

SEC

AD-A218 269

IMPLEMENTATION PAGE

Form Approved
OMB No. 0704-0188

DTIC FILE COPY



| | | | |
|---|--|---|--------------------------------------|
| 1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED | | 1b. RESTRICTIVE MARKINGS NONE | |
| 2a. SECURITY CLASSIFICATION AUTHORITY | | 3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. | |
| 2b. DECLASSIFICATION/DOWNGRADING SCHEDULE | | | |
| 4. PERFORMING ORGANIZATION REPORT NUMBER(S) | | 5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-012 | |
| 6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT UNIV OF NORTH CAROLINA AT CHAPEL HILL | 6b. OFFICE SYMBOL (If applicable) | 7a. NAME OF MONITORING ORGANIZATION AFIT/CIA | |
| 6c. ADDRESS (City, State, and ZIP Code) | | 7b. ADDRESS (City, State, and ZIP Code) Wright-Patterson AFB OH 45433-6583 | |
| 8a. NAME OF FUNDING/SPONSORING ORGANIZATION | 8b. OFFICE SYMBOL (If applicable) | 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | |
| 8c. ADDRESS (City, State, and ZIP Code) | | 10. SOURCE OF FUNDING NUMBERS | |
| | | PROGRAM ELEMENT NO. | PROJECT NO. |
| | | TASK NO. | WORK UNIT ACCESSION NO. |
| 11. TITLE (Include Security Classification) (UNCLASSIFIED) An Algebraic Language for Query and Update of Temporal Databases | | | |
| 12. PERSONAL AUTHOR(S) Leslie Edwin McKenzie, Jr. | | | |
| 13a. TYPE OF REPORT XPERIMENT/ DISSERTATION | 13b. TIME COVERED FROM _____ TO _____ | 14. DATE OF REPORT (Year, Month, Day) 1988 | 15. PAGE COUNT 315 |
| 16. SUPPLEMENTARY NOTATION APPROVED FOR PUBLIC RELEASE IAW AFR 190-1 ERNEST A. HAYGOOD, 1st Lt, USAF Executive Officer, Civilian Institution Programs | | | |
| 17. COSATI CODES | | 18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) | |
| FIELD | GROUP | SUB-GROUP | |
| | | | |
| 19. ABSTRACT (Continue on reverse if necessary and identify by block number) | | | |
| 20. DISTRIBUTION/AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS | | 21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED | |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF | | 22b. TELEPHONE (Include Area Code) (513) 255-2259 | 22c. OFFICE SYMBOL AFIT/CI |

D-TIC
ELECTED
FEB 15 1990
S D D

90 02 14 042

An Algebraic Language for Query and Update of Temporal Databases

by

Leslie Edwin McKenzie, Jr.

A Dissertation submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1988

| | |
|---------------------|-------------------------------------|
| Accession For | |
| NTIS CRA&I | <input checked="" type="checkbox"/> |
| DTIC TAB | <input type="checkbox"/> |
| Unannounced | <input type="checkbox"/> |
| Justification | |
| By | |
| Distribution / | |
| Availability Codes | |
| Dist | Avail and/or Special |
| A-1 | |

Approved by:

Richard Snodgrass

Advisor - Dr. Richard Snodgrass

John B. Smith

Reader - Dr. John B. Smith

J. Dean Brock

Reader - Dr. J. Dean Brock

**©1988
Leslie E. Mckenzie, Jr.
ALL RIGHTS RESERVED**

(A)

LESLIE EDWIN MCKENZIE, JR. An Algebraic Language for Query and Update of Temporal Databases (Under the direction of RICHARD SNODGRASS.)

Abstract

Although time is a property of events and objects in the real world, conventional relational database management systems (RDBMS's) can't model the evolution of either the objects being modeled or the database itself. Relational databases can be viewed as *snapshot* databases in that they record only the current database state, which represents the state of the enterprise being modeled at some particular time. We extend the relational algebra to support two orthogonal aspects of time: *valid time*, which concerns the modeling of time-varying reality, and *transaction time*, which concerns the recording of information in databases. In so doing, we define an algebraic language for query and update of temporal databases.

The relational algebra is first extended to support valid time. Historical versions of nine relational operators (i.e., union, difference, cartesian product, selection, projection, intersection, Θ -join, natural join, and quotient) are defined and three new operators (i.e., historical derivation, non-unique aggregation, and unique aggregation) are introduced. Both the relational algebra and this new historical algebra are then encapsulated within a language of commands to support transaction time. The language's semantics is formalized using denotational semantics. Rollback operators are added to the algebras to allow relations to be rolled back in time. The language accommodates scheme and contents evolution, handles single-command and multiple-command transactions, and supports queries on valid time. The language is shown to have the expressive power of the temporal query language TQuel. (KR) ←

The language supports both unmaterialized and materialized views and accommodates a spectrum of view maintenance strategies, including incremental, recomputed, and immediate view materialization. Incremental versions of the snapshot and historical operators are defined to support incremental view materialization. A prototype query processor was built for TQuel to study incremental view materialization in temporal databases. Problems that arise when materialized views are maintained incrementally are discussed, and solutions to those problems are proposed.

Criteria for evaluating temporal algebras are presented. Incompatibilities among the criteria are identified and a maximal set of compatible evaluation criteria is proposed. Our language and other previously proposed temporal extensions of the relational algebra are evaluated against these criteria.

Acknowledgements

My advisor, Rick Snodgrass, deserves much of the credit for this work. He encouraged me to do research in temporal databases and was a constant source of good ideas, inspiration, and support. His guidance made this research possible. I also am indebted to the other members of my committee, Jay Nievergelt, Jan Prins, and especially Dean Brock and John Smith, for their review of the work and suggested improvements.

I would like to thank the faculty, staff, and students at the University of North Carolina for creating an environment in which I could work on such an interesting and challenging idea. The experience has been both rewarding and enjoyable. I especially would like to thank Bharat Jayaraman and Peter Mills for their review of, and suggested improvements to, portions of this document, and Pamela Payne for her administrative support. I also would like to thank my fellow students, Teresa Thomas, Will Partain, Phil Amburn, Sundar Varadarajan, Ralph Cook, Shie-Jue Lee, and John Cromer, for their help and encouragement.

Finally, I would like to thank the United States Air Force for its financial support and my family for their continued love and encouragement throughout this process.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Terminology | 3 |
| 1.1.1 | Conceptual Models of Time | 3 |
| 1.1.2 | Taxonomy of Time in Databases | 4 |
| 1.2 | The Problem | 7 |
| 1.3 | The Approach | 9 |
| 1.4 | Scope of Research | 12 |
| 1.5 | Structure of the Dissertation | 12 |
| 1.6 | Notational Conventions | 14 |
| 2 | Previous Work | 15 |
| 2.1 | Temporal Query Languages | 15 |
| 2.2 | Algebras | 17 |
| 2.3 | Storage Structures and Access Strategies | 17 |
| 2.4 | Strategies for Efficient Query Processing | 19 |
| 3 | Supporting Valid Time: A Historical Algebra | 21 |
| 3.1 | Approach | 21 |
| 3.2 | Historical Relation | 23 |
| 3.3 | Historical Operators | 25 |
| 3.3.1 | Union | 26 |
| 3.3.2 | Difference | 27 |
| 3.3.3 | Cartesian Product | 27 |
| 3.3.4 | Selection | 28 |
| 3.3.5 | Projection | 29 |

| | | |
|----------|--|-----------|
| 3.3.6 | Historical Derivation | 32 |
| 3.4 | Aggregates | 36 |
| 3.4.1 | Partitioning Function | 38 |
| 3.4.2 | Non-unique Aggregates | 40 |
| 3.4.3 | Unique Aggregates | 42 |
| 3.4.4 | Expressions in Aggregates | 43 |
| 3.5 | Preservation of the Value-equivalence Property | 44 |
| 3.6 | Additional Aspects of the Algebra | 46 |
| 3.7 | Summary | 50 |
| 4 | Adding Transaction Time | 52 |
| 4.1 | Approach | 53 |
| 4.2 | The Language | 56 |
| 4.2.1 | Syntax | 56 |
| 4.2.2 | Semantic Domains | 60 |
| 4.2.3 | A Semantic Type System for Expressions | 63 |
| 4.2.4 | Expressions | 71 |
| 4.2.5 | Commands | 77 |
| 4.2.6 | Programs | 90 |
| 4.2.7 | Language Properties | 91 |
| 4.3 | Additional Aspects of the Rollback Operators | 94 |
| 4.4 | Summary and Related Work | 95 |
| 5 | Equivalence With TQuel | 99 |
| 5.1 | TQuel Database | 99 |
| 5.2 | TQuel Retrieve Statement | 103 |
| 5.2.1 | Semantics | 104 |
| 5.2.2 | Correspondence Theorem | 107 |
| 5.3 | TQuel Aggregates | 112 |
| 5.3.1 | Aggregate Functions | 113 |
| 5.3.2 | In the Target List | 117 |
| 5.3.3 | In the Inner Where Clause | 120 |
| 5.3.4 | In the Inner When Clause | 122 |

| | |
|---|------------|
| 5.3.5 In the Outer Where Clause | 123 |
| 5.3.6 In the Outer When Clause | 124 |
| 5.3.7 Multiply-nested Aggregation | 124 |
| 5.3.8 Correspondence Theorem | 124 |
| 5.4 TQuel Modification Statements | 125 |
| 5.4.1 Create Statement | 125 |
| 5.4.2 Append Statement | 126 |
| 5.4.3 Delete Statement | 127 |
| 5.4.4 Replace Statement | 128 |
| 5.4.5 Destroy Statement | 129 |
| 5.4.6 Correspondence Theorem | 130 |
| 5.5 Language Correspondence | 130 |
| 5.6 Summary | 131 |
| 6 Adding Support for Views | 132 |
| 6.1 Background | 133 |
| 6.2 Approach | 136 |
| 6.3 Incremental Snapshot Algebra | 138 |
| 6.3.1 Snapshot Differential | 138 |
| 6.3.2 Incremental Snapshot Operators | 140 |
| 6.4 Incremental Historical Algebra | 143 |
| 6.4.1 Historical Differential | 143 |
| 6.4.2 Historical Operators | 146 |
| 6.5 Language Extensions | 153 |
| 6.5.1 Syntax | 153 |
| 6.5.2 Semantic Domains | 154 |
| 6.5.3 Type System | 154 |
| 6.5.4 Expressions | 155 |
| 6.5.5 Commands | 158 |
| 6.6 Scheme Evolution in the Presence of Views | 165 |
| 6.7 Summary | 166 |

| | |
|---|------------|
| 7 Incremental View Materialization | 168 |
| 7.1 Background | 168 |
| 7.2 Approach | 174 |
| 7.3 Architecture | 175 |
| 7.4 TQuel Prototype | 179 |
| 7.4.1 The Code Generator | 180 |
| 7.4.2 Interpreter | 182 |
| 7.5 Implementation Issues | 182 |
| 7.5.1 Query Optimization | 182 |
| 7.5.2 Local Storage Strategies at Operator Nodes | 187 |
| 7.5.3 Representation of Attribute Time-stamps | 191 |
| 7.5.4 Representation of Historical Differentials | 191 |
| 7.5.5 Local Processing Strategies at Operator Nodes | 193 |
| 7.5.6 Dynamic Time-stamps | 197 |
| 7.5.7 Deferred View Materialization | 199 |
| 7.5.8 Concurrency Control and Recovery | 200 |
| 7.5.9 Aggregates | 204 |
| 7.6 Summary | 204 |
| 8 Evaluation Criteria | 206 |
| 8.1 Temporal Extensions of the Snapshot Algebra | 207 |
| 8.2 Criteria | 217 |
| 8.3 Properties not Included as Criteria | 228 |
| 8.4 Incompatibilities | 230 |
| 8.5 An Evaluation of Historical and Temporal Algebras | 234 |
| 8.5.1 Conflicting Criteria | 235 |
| 8.5.2 Compatible Criteria | 241 |
| 8.5.3 Evaluation Summary | 246 |
| 8.6 Review of Design Decisions | 248 |
| 8.6.1 Time-stamped Attributes | 248 |
| 8.6.2 Set-valued Time-stamps | 248 |
| 8.6.3 Single-valued Attributes | 248 |

| | | |
|----------|--|------------|
| 8.6.4 | Extended Operator Semantics | 249 |
| 8.6.5 | New Temporal Operators | 249 |
| 8.6.6 | Transaction Time and Relation States | 249 |
| 8.7 | Summary | 250 |
| 9 | Conclusions and Future Work | 251 |
| 9.1 | Contributions | 251 |
| 9.1.1 | Language | 251 |
| 9.1.2 | Temporal Algebra | 252 |
| 9.1.3 | Incremental Temporal Algebra | 253 |
| 9.1.4 | Prototype Implementation | 253 |
| 9.1.5 | Evaluation Criteria | 253 |
| 9.2 | Conclusions | 254 |
| 9.3 | Future Work | 256 |
| | Bibliography | 258 |
| A | Symbols | 272 |
| B | Auxiliary Functions | 276 |
| B.1 | Semantic Functions | 276 |
| B.2 | Other Auxiliary Functions | 292 |
| C | Language Syntax | 306 |
| C.1 | Syntax | 306 |
| C.2 | Extensions | 310 |
| | Index | 312 |

List of Figures

| | |
|--|-----|
| 1.1 Snapshot Relation | 5 |
| 1.2 Rollback Relation | 6 |
| 1.3 Historical Relation | 6 |
| 1.4 Temporal Relation | 7 |
| 6.1 View Dependency Graph for Base Relation S | 133 |
| 6.2 Classification of Relations by Type and View Maintenance Strategy | 135 |
| 7.1 Parse Tree for View S3 | 169 |
| 7.2 Update Network for View S3 As Formalized by Horwitz and Snodgrass | 170 |
| 7.3 Update Network for View S3 As Formalized by Roussopoulos | 171 |
| 7.4 Conventional Architecture for Query Processing | 176 |
| 7.5 Extended Architecture for Query Processing | 177 |
| 7.6 View Update Network for View S3 | 178 |
| 7.7 Database Update Network | 179 |
| 7.8 Update Network for a TQuel View | 181 |
| 7.9 Model of Concurrency Control and Recovery [Bernstein et al. 1987] | 201 |
| 8.1 Outline of Equivalence Proof | 221 |
| 8.2 Outline of Reduction Proof | 224 |
| 8.3 Historical Relation | 225 |
| 8.4 $A \dot{\times} (B - C)$ and $(A \dot{\times} B) - (A \dot{\times} C)$ | 232 |
| 8.5 Conceptual View of the Difference Operator Applied to Historical Relations | 239 |
| 8.6 Cartesian Product of Historical Relations | 240 |

List of Tables

| | |
|---|-----|
| 4.1 Define Relation Command | 80 |
| 4.2 Modify Relation Command | 83 |
| 7.1 Time Complexity of Incremental Historical Operators | 198 |
| 8.1 Representation of Time in the Algebras | 217 |
| 8.2 Objects and Operations in the Algebras | 218 |
| 8.3 Criteria for Evaluating Temporal Extensions of the Snapshot Algebra . . . | 220 |
| 8.4 Incompatibilities Among Criteria | 234 |
| 8.5 Evaluation of Temporal Algebras Against Criteria | 236 |
| 8.5 Evaluation of Temporal Algebras Against Criteria (cont'd) | 237 |
| 8.6 Classification of Algebras According to Criteria Satisfied | 246 |

Chapter 1

Introduction

Time is a property of both events and objects in the real world. Events occur at specific points in time; objects and the relationships among objects exist over time. The ability to model this temporal aspect of real-world phenomena is essential to many computer system applications (e.g., econometrics, banking, inventory control, medical records, airline reservations, personnel records). Although techniques for encoding time-varying information in conventional databases have been developed in many application areas, these techniques are necessarily ad hoc and application-specific. They are not supported by a formal data model.

Conventional database management systems (DBMS's), in general, provide no direct support for time. This lack of support for time limits the effectiveness of conventional databases as accurate models of reality for the following three reasons.

- Conventional databases don't model time-varying aspects of real-world phenomena. They record the state of the enterprise being modeled at some particular time, but not the evolution of the enterprise over time. Hence, DBMS's support only queries that can be answered on a single recorded state of the enterprise; they don't support queries that require knowledge of the enterprise's history. They also don't allow either *retroactive* changes or *postactive* changes (i.e., changes that will occur in the future) [Snodgrass & Ahn 1985] to the enterprise to be recorded in the database.
- A database itself changes state when it is updated. In conventional DBMS's, however, out-of-date information is discarded when a database is updated; past database states aren't retained for future reference. Hence, DBMS's allow queries to be evaluated only on the current database state; they don't allow the database to be rolled back in time for query evaluation on a past database state.
- Conventional DBMS's don't distinguish between the state of the database and the state of the enterprise being modeled. DBMS's record only the current database state, which is assumed to represent the current state of the enterprise being modeled. The current database state, however, may not be, and often will not be, consistent with the

current state of the enterprise being modeled, simply because of delays and errors in recording changes to the enterprise's state. DBMS's provide no facilities for recording retroactively these periods of inconsistency.

EXAMPLE. Consider a simple course enrollment database at a university. Assume that Phil, on September 1, enrolls in a mathematics course effective on September 2 but his enrollment in this course is not recorded in the database until September 3. Hence, on September 2 the database is inconsistent with the enrollment in this course, and queries on the database on that day may produce erroneous results. Once the database is updated on September 4, the inconsistency is resolved. The database, however, contains no record of either Phil's enrollment in this mathematics course before September 4 or the inconsistency that existed on September 3. Furthermore, the database state before this latest change no longer exists. Because queries are always evaluated on the current database state, which is assumed to represent the current state of the enterprise being modeled, neither queries concerning Phil's enrollment before September 4 nor queries on a database state before September 4 are allowed. The query "What are the courses in which Phil is enrolled?" can be answered, but the query "When did Phil enroll in Math?" can't be answered because Phil's enrollment history is not stored in the database. \square

The need for direct database support for time has received increasing attention recently. Over the past decade researchers in disciplines as varied as artificial intelligence, logic, natural language processing, distributed processing, and database systems have studied the role that time plays in information processing. Bibliographies [Bolour et al. 1982, McKenzie 1986, Stam & Snodgrass 1988] show that the number of works relating time to information processing has increased exponentially during the last few years. One area of continuing research interest is the development of a *temporal data model* capable of supporting the temporal aspects of both real-world phenomena and the databases that model these phenomena. The primary focus of research in this area has been extending the relational data model [Codd 1970] to support time-varying information.

In this dissertation we add support for time to one component of the relational data model: the relational algebra. The relational algebra is an important component of the relational data model because it can serve as the underlying evaluation mechanism for queries in user-oriented, high-level query languages such as Quel and SQL [Ullman 1982]. We extend the relational algebra to support the orthogonal aspects of time that concern the modeling of time-varying reality and the recording of information in a database. In so doing, we define an algebraic language for query and update of temporal databases. This language can be used as the underlying evaluation mechanism for queries in temporal query languages such as TQuel [Snodgrass 1987]. The language also is a solution to the time-related problems of conventional DBMS's described above. It can be used to record the evolution of an enterprise over time for both retrieval and update, retain all past states of the database, and distinguish between the state of the database and the state of the enterprise being modeled. Hence, queries that require knowledge of the history of the enterprise being modeled can be supported, and queries can be evaluated on either the current or any past database state. Also, both retroactive and postactive changes to the

enterprise can be recorded in the database, as can periods when the states of the database and enterprise are known to have been inconsistent.

In the next section, we introduce some basic terminology. Then, we identify specific problems in extending the relational algebra to handle time directly, describe our approach for solving these problems, and discuss the scope of our research. We conclude this chapter with an overview of the rest of the dissertation.

1.1 Terminology

A *database* is a set of structured data that models some aspect of a real-world enterprise or phenomenon (e.g., a company's information about its employees). A database's *scheme* describes the database's structure; the database's contents must adhere to that structure [Date 1976, Ullman 1982]. A *database management system (DBMS)* is the system used by persons to access and manipulate the data stored in a database. Most DBMS's are based on either the relational, network, or hierarchical data model [Ullman 1982]. In this dissertation, we consider only the relational data model.

1.1.1 Conceptual Models of Time

Two basic conceptual time models have been proposed: the *continuous model*, in which time is viewed as being isomorphic to the real numbers, and the *discrete model*, in which time is viewed as being isomorphic to the natural numbers (or a discrete subset of the real numbers) [Clifford & Tansel 1985]. In the continuous model, each real number corresponds to a "point" in time, whereas in the discrete model, each natural number corresponds to a non-decomposable unit of time having an arbitrary duration. Although the two time models represent time differently, they share one important property; they both require that time be ordered linearly. Hence, for two non-equal times, t_1 and t_2 , either t_1 is "before" t_2 or t_2 is "before" t_1 [Anderson 1982, Clifford & Tansel 1985].

"Instant," [Gadia 1986] "moment," [Allen & Hayes 1985] "time quantum," [Anderson 1982] and "time unit" [Navathe & Ahmed 1986, Tansel 1986] are just some of the terms used in the literature to describe a non-decomposable unit of time in the discrete model. To avoid confusion between a point in the continuous model and a non-decomposable unit of time in the discrete model, we refer to a non-decomposable unit of time in the discrete model as a *chronon* [Ariav 1986] and define an *interval* to be a set of consecutive chronons. Although the duration of each chronon in a set of times need not be the same, the duration of a chronon is usually fixed by the granularity of the measure of time being used (e.g., day, week, hour, second). A chronon typically is denoted by an integer, corresponding to a single granularity, but may also be denoted by a sequence of integers, corresponding to a nested granularity. For example, if we assume a granularity of a day relative to January 1, 1980, then the integer 1901 denotes March 15, 1985. If, however we assume a nested granularity of (year, month, day), then the sequence (6, 3, 15) denotes March 15, 1985.

We use the discrete model in this dissertation. Several practical arguments are given in the literature that support our preference for the discrete model over the continuous model. First, measures of time are inherently imprecise [Anderson 1982, Clifford & Tansel 1985]. Clocking instruments invariably report the occurrence of events in terms of chronons, not time "points." Hence, events, even so-called "instantaneous" events, can at best be measured as having occurred during a chronon. Secondly, most natural language references to time are compatible with the discrete time model. For example, when we say that an event occurred at 4:30 p.m., we usually don't mean that the event occurred at the "point" in time associated with 4:30 p.m., but at some time in the chronon (minute) associated with 4:30 p.m. [Anderson 1982]. Thirdly, the concepts of chronon and interval allow us to model naturally events that are not instantaneous, but have duration [Anderson 1982]. Finally, any implementation of a data model with a temporal dimension will of necessity have to have some discrete encoding for time [Snodgrass 1987].

1.1.2 Taxonomy of Time in Databases

There are three orthogonal aspects of time that a relational database management system (RDBMS) needs to support: valid time, transaction time, and user-defined time [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. *Valid time* concerns modeling time-varying reality. The valid time of, say, an event is the clock time when the event occurred in the real world, independent of the recording of that event in some database. Other terms found in the literature that have a similar meaning include intrinsic time [Bubenko 1977], effective time [Ben-Zvi 1982], and logical time [Dadam et al. 1984, Lum et al. 1984]. *Transaction time*, on the other hand, concerns the storage of information in the database. The transaction time of an event is the transaction number (an integer) of the transaction that stored the information about the event in the database. Other terms found in the literature that have a similar meaning include extrinsic time [Bubenko 1977], registration time [Ben-Zvi 1982], and physical time [Dadam et al. 1984, Lum et al. 1984]. *User-defined time* is an uninterpreted domain for which the RDBMS supports the operations of input, output, and perhaps comparison. As its name implies, the semantics of user-defined time is provided by the user or application program. These three aspects of time are orthogonal in the support required of the RDBMS. User-defined time is supported by the relational algebra, in that it is simply another domain, such as integer or character string, provided by the RDBMS [Bontempo 1983, Overmyer & Stonebraker 1982, Tandem 1983]; valid time and transaction time, however, are not supported.

Valid time, unlike transaction time, is a multifaceted aspect of time. Different times may be used in defining the existence of a single object or relationship (e.g., the time a student completes degree requirements and the time of the student's graduation ceremony may both be used in specifying the student's graduation from college). Also, the properties of an object or relationship all need not change at the same time (e.g., an employee's promotion may, but need not, be accompanied by a change in salary or address). We consider a single, but arbitrary, concept of valid time.

Relations may be classified, depending on their support for valid time and transaction

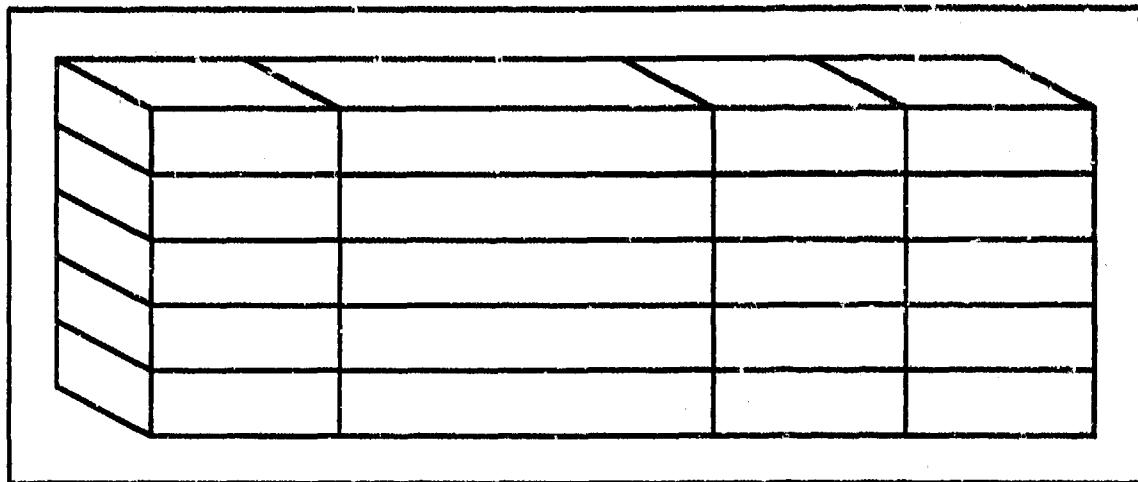


Figure 1.1: Snapshot Relation

time, as either snapshot, rollback, historical, or temporal relations [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. *Snapshot relations* support neither valid time nor transaction time. They model an enterprise at one particular point in time. As a snapshot relation is changed to reflect changes in the enterprise being modeled, past states of the relation, representing past states of the enterprise, are discarded. A snapshot relation consists of a set of *tuples* with the same set of *attributes*, and is usually represented as a two-dimensional table with attributes as columns and tuples as rows, as shown in Figure 1.1. Note that snapshot relations are exactly those relations supported by the relational algebra. Hence, for clarity, we will refer to the relational algebra hereafter as the *snapshot algebra*. *Rollback relations* support transaction time but do not support valid time. They may be represented as a sequence of snapshot states indexed by transaction time, as shown in Figure 1.2. (Here, the last transaction deleted one tuple and appended another.) Because they record the history of database activity, rollback relations can be rolled back to one of their past snapshot states for querying, hence their name.

Historical relations support valid time but do not support transaction time. They model the history, as it is best known, of an enterprise. When a historical relation is changed, however, its past state, like that of a snapshot relation, is discarded. A historical relation may be represented as a three-dimensional solid, as shown in Figure 1.3. Because they record the history of the enterprise being modeled, historical relations support historical queries. They do not, however, support rollback operations. *Temporal relations* support both valid time and transaction time. They may be represented as a sequence of historical states indexed by transaction time, as shown in Figure 1.4. Because they record both the history of the enterprise being modeled and the history of database activities, temporal relations support both historical queries and rollback operations.

Data models that support these four classes of relations have several important properties. First, a relation's scheme can no longer be defined in terms of the relation's attributes

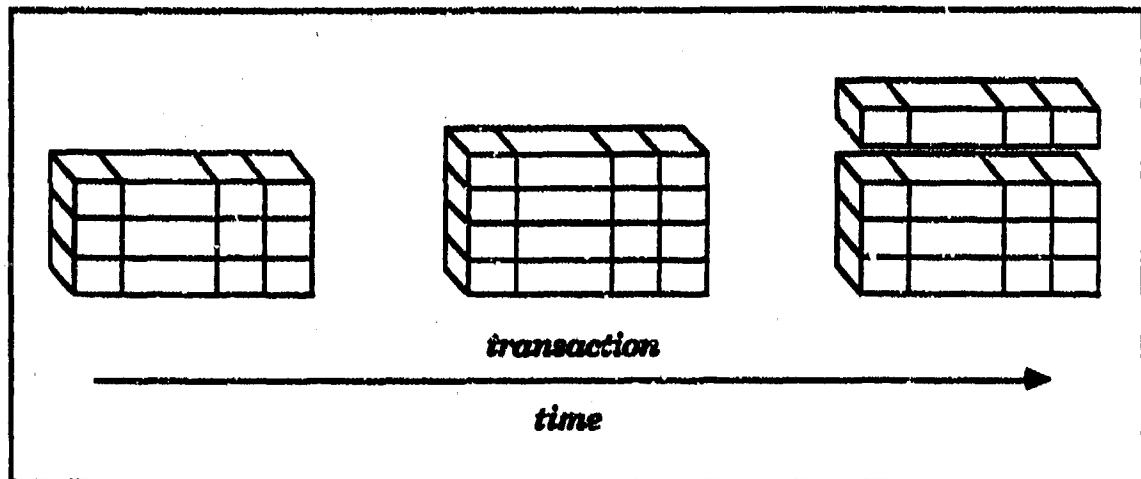


Figure 1.2: Rollback Relation

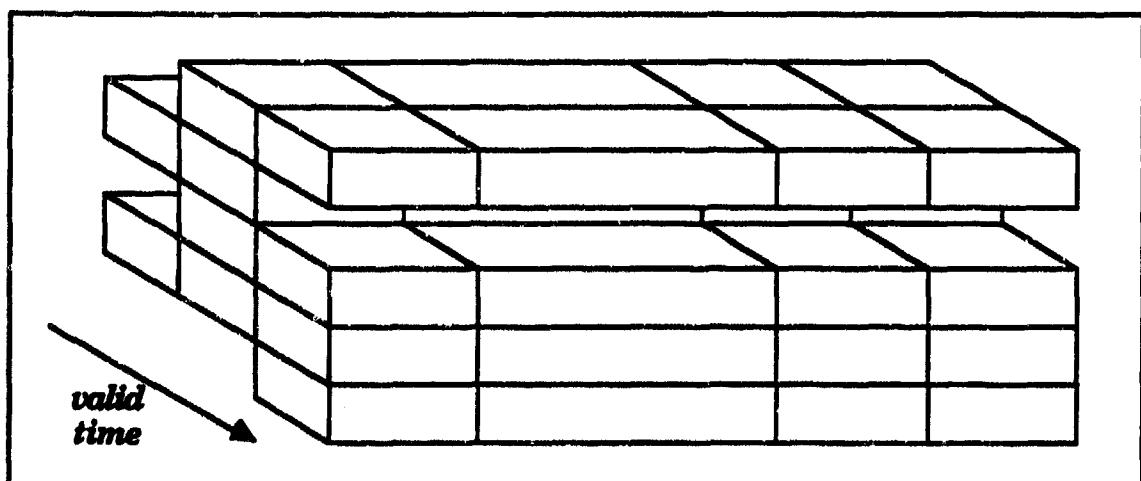


Figure 1.3: Historical Relation

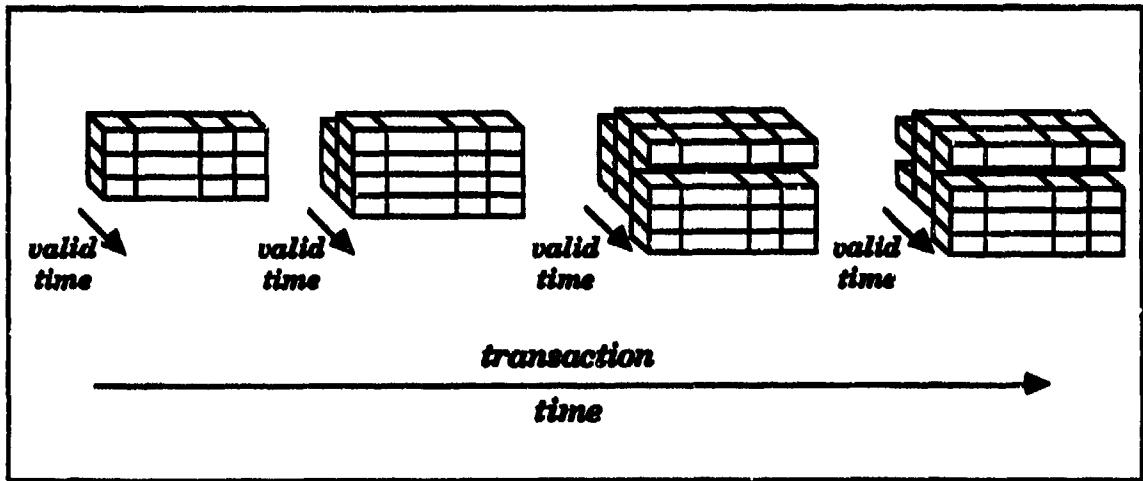


Figure 1.4: Temporal Relation

alone; it must also include the relation's *class* (i.e., snapshot, rollback, historical, or temporal). Second, rollback and temporal relations, unlike snapshot and historical relations, are *append-only* relations. Information, once added to a rollback or temporal relation, cannot be deleted; otherwise, rollback operations could not be supported. Third, rollback and temporal relations must record the evolution of their schemes as well as their contents, as both may change over time. Fourth, valid time and transaction time are orthogonal aspects of time. A relation may support either valid time or transaction time without supporting both. Also, the time when an enterprise changes (i.e., valid time) need not be, and usually will not be, the same as the time when the database is updated (i.e., transaction time) to reflect that change. Finally, the same measures of time need not be used for valid time and for transaction time. For example, a temporal relation will have a variable granularity, which changes with each update, for transaction time but could have a fixed granularity (e.g., second) for valid time.

1.2 The Problem

A *query* in a RDBMS is a computation that *derives* a relation from one or more underlying *base relations* or *views*, where a view is simply a relation defined, via an algebraic expression, by a function on other relations in the database (c.f., Chapter 6). A *query system* is a formal system for expressing queries [Maier 1983]. There are three principal query systems for the relational model: tuple predicate calculus, domain predicate calculus, and the snapshot algebra [Ullman 1982]. These query systems were proposed by [Codd 1972] and are equivalent in expressive power. The calculi are non-procedural; they specify what the result of a query should be without specifying how it is to be derived. Hence, the calculi are useful in defining high-level, non-procedural query languages for RDBMS's. The algebra, however, is procedural; it specifies what the result of a query should be and the method

to be used in its derivation. Hence, the algebra is useful in implementing query processors for RDBMS's. Expressions in the snapshot algebra can be defined in terms of relations and only five operators: union, set difference, cartesian product, projection, and selection [Ullman 1982]. Because the algebra is simply a system for writing expressions that evaluate to derived relations, it is also useful in implementing other aspects of RDBMS's, including update operations, database views, and integrity constraints [Date 1986A].

The snapshot algebra supports only snapshot relations. Although a snapshot relation is time-varying in that it changes over time due to the insertion, deletion, and modification of tuples, no record of the evolution of either the relation or the enterprise that it models is maintained. Hence, the relational data model cannot handle historical queries or queries whose frame of reference is other than the present. If we assume that snapshot relations are always maintained "up-to-date," then the relational data model can only answer questions of the form "What do you know about the present state of an enterprise as of now?" Questions of the form "What do (did) you know about the history of an enterprise as of now (as of some time in the past)?" cannot be answered. The broader class of queries represented by this later question can only be answered if both valid time and transaction time are added to the algebra.

Over the past decade, no less than 10 proposals [Ben-Zvi 1982, Clifford & Croker 1987, Gadia 1988, Gadia & Yeung 1988, Jones et al. 1979, Lorentzos & Johnson 1987A, Navathe & Ahmed 1986, Sadeghi 1987, Sarda 1988, Tansel 1986] for extending the snapshot algebra to include one or more aspects of time have appeared in the literature. All but two can be termed *historical algebras* because they address only the problem of adding valid time to the snapshot algebra. Ben-Zvi addresses the problem of adding both valid time and transaction time to the snapshot algebra [Ben-Zvi 1982]. He defines formally an extension of the snapshot algebra that includes valid time and one aspect of transaction time (contents evolution). He also describes an approach for handling scheme evolution. He does not, however, define a unified approach for handling valid time and both contents and scheme evolution; the retrieval and update semantics of his model account for valid time and contents evolution, but not scheme evolution. Gadia and Yeung also address the problem of adding both valid time and transaction time to the snapshot algebra [Gadia & Yeung 1988]. They propose that transaction time be treated as one dimension of a multi-dimensional time-stamp, whose other dimensions record various facets of valid time. They do not, however, define update semantics for this model. A formally defined extension of the snapshot algebra that includes valid time and both aspects of transaction time has yet to be proposed.

Although several temporal extensions of the snapshot algebra have been proposed, criteria for evaluating the relative merit of these extensions have been left largely unexplored. The focus of research has been definition of algebras that include some aspect of time rather than identification of properties that these new algebras should have. A comprehensive set of well-defined, objective criteria for evaluating temporal extensions of the snapshot algebra has yet to be defined. Without such a set of criteria, evaluation and comparison of the proposed algebras is impossible.

Implementation of a temporal extension of the snapshot algebra as the evaluation mechanism for queries in a temporal database management system (TDBMS's) is another subject that has received only limited research attention. A formally defined algebra that includes both valid time and transaction time and satisfies a maximal subset of evaluation criteria, while of theoretical importance, is likely to have little practical use in TDBMS's if it cannot be implemented at reasonable cost. How best to implement a temporal extension of the snapshot algebra for query processing in a TDBMS is a question that has yet to be answered.

1.3 The Approach

The basic goal of our research was extension of the snapshot algebra to support both valid time and transaction time. Because valid time and transaction time are orthogonal, we were able to deal with each of the two aspects of time in isolation. The research itself was conducted in three phases. In the first phase, we extended the snapshot algebra to support valid time by defining a historical algebra. We then encapsulated both the snapshot algebra and this new historical algebra in a language of commands to handle both aspects of transaction time: scheme evolution and contents evolution. Finally, we extended the language to accommodate views and defined an architecture for query processing in TDBMS's that accommodates the incremental maintenance of materialized views. Incrementally maintained materialized views are important to our research because they may be used to implement certain classes of recurring queries efficiently [Hanson 1987A, Roussopoulos 1987]. The result of our research is a formally defined algebraic language for database query and update that

- Includes valid time and both aspects (scheme evolution and contents evolution) of transaction time;
- Supports snapshot, rollback, historical, and temporal relations;
- Satisfies a maximal subset of evaluation criteria; and,
- Serves as the underlying evaluation mechanism for historical query processing in TDBMS's.

We next describe the specific contributions each phase of our research made to this language.

Our primary objective in extending the snapshot algebra to include valid time and transaction time was to define an algebraic language for query and update of temporal databases that satisfies a maximal subset of evaluation criteria. We identified 29 criteria for evaluating such languages. The criteria are restricted to those properties that are well-defined, have an objective basis for being evaluated, and are arguably beneficial. As all the evaluation criteria are not compatible, we also defined a maximal subset of these criteria.

We then considered this maximal subset of evaluation criteria when extending the snapshot algebra to support valid time. All design decisions were made so that the resulting algebra would possess as many of the most desirable properties of historical algebras as possible. Definition of the historical algebra thus satisfied our goal to extend the snapshot algebra to include valid time, support historical relations, and satisfy a maximal subset of evaluation criteria.

A relation is defined by its scheme and contents. Database transactions change one or more relations by changing either their contents or both their schemes and their contents. When snapshot and historical relations are changed, their old scheme and contents can be discarded. When rollback and temporal relations are changed, however, their old scheme and contents must be retained. Hence, the fundamental problem that must be solved in extending the snapshot and historical algebras to include transaction time is how best to model the evolution of a relation's scheme and contents so that past states of rollback and temporal relations are accessible.

An algebra by definition is side-effect-free, but the essential aspect of a database transaction is solely its side-effect of modifying the database. One awkward but perhaps feasible solution would have been to add the database as a parameter to every operator in the snapshot and historical algebras. We adopted a different strategy, leaving the basic structure of the algebras intact, and instead encapsulating them in another structure of *commands* that provide the needed side-effects. We first added a new algebraic operator called *rollback* to both the snapshot and historical algebras to make past states of rollback and temporal relations available in the algebras, respectively. Fortunately, the rollback operation is side-effect-free, so it was easily incorporated into the algebras. We then defined commands that modify a relation's scheme and contents. For completeness, we also defined commands that modify a relation's class (i.e., snapshot, historical, rollback, or temporal). We used denotational semantics to define the semantics of commands, due to its success in formalizing operations involving side-effects, such as assignment, in programming languages [Gordon 1979, Stoy 1977]. Hence, we extended the snapshot and historical algebras to include transaction time by extending the algebras to include a rollback operator and defining a *language* for database update, with the slightly extended algebras as significant components. Definition of this language satisfied our goal to extend the snapshot algebra to include both aspects of transaction time and to support rollback and temporal relations.

Our extension of the snapshot algebra to support valid and transaction time will be useful as the underlying model for TDBMS's only if it can be implemented at reasonable cost. Because several studies have already identified appropriate storage structures and access strategies for rollback, historical, and temporal relations and because our approach for adding valid time and transaction time to the snapshot algebra is compatible with those storage structures and access strategies, we only considered the appropriateness of our extension of the snapshot algebra as the evaluation mechanism for queries in a TDBMS.

Queries in TDBMS's can be grouped into three broad classes: snapshot queries, rollback queries, and non-rollback, historical queries. Snapshot queries involve neither valid nor transaction time; Ahn has shown that this class of queries can be supported in TDBMS's

without performance penalty if appropriate storage structures are used [Ahn 1986A]. Rollback queries, which reference either rollback or temporal relations, are queries asked "as of" some time in the past. Because the past states of rollback and temporal relations never change, both the cost and the result of processing a rollback query are constant over time. If a rollback query's execution frequency is sufficiently high, it is cost-effective to evaluate the query once and cache the result for future reference. Otherwise, it is cost-effective to simply re-evaluate the query each time it is asked. Historical queries are queries on the current state of historical and temporal relations. Because the size of the current state of historical and temporal relations is likely to increase monotonically over time, the cost of evaluating a given historical query is also likely to increase monotonically over time. Furthermore, as only the most recent historical data in the current state of a historical or temporal relation is likely to change between accesses, there is likely to be an increasing amount of redundant processing associated with each repeated evaluation of a historical query. Application-specific factors such as the frequency of query evaluation, update patterns, the cost of each evaluation, and the cost of alternate query processing techniques determine whether re-evaluation of a recurring historical query each time it is asked is cost-effective. Yet, there will be a subclass of recurring historical queries in many applications for which query re-evaluation each time a query is asked will have unacceptable cost. Also, the size of this subclass of recurring historical queries will increase during the life of a temporal database.

We propose that incrementally maintained materialized views be used to implement recurring historical queries for which query re-evaluation has an unacceptable cost. Under this proposal, the result of a recurring historical query would be cached as a materialized view and changed incrementally to reflect updates to the query's underlying relations. Cacheing the results of recurring queries as incrementally maintained materialized views has been shown to be more efficient than query re-evaluation for evaluating recurring non-temporal queries, and sometimes significantly so, if the execution frequency of the queries is sufficiently high, the sizes of the queries' underlying relations are sufficiently large, and the volatility of the queries' underlying relations, defined as the percentage of tuples that change between accesses, is sufficiently low [Hanson 1987A, Horwitz 1986, Roussopoulos 1987]. Incremental view materialization will be applicable to an even larger subclass of recurring historical queries, as the cost of evaluating a historical query is typically greater than the cost of evaluating an analogous non-temporal query.

To support incremental view materialization in TDBMS's, we defined incremental versions of the snapshot and historical algebras and defined an architecture for incremental view materialization in which nodes in query plans correspond to operators in the incremental algebras (c.f., Chapter 7). We surveyed various techniques developed for efficient implementation of both incremental and non-incremental query processors in RDBMS's and analyzed their applicability to our architecture for historical query processing in TDBMS's. We considered only implementation issues, such as query optimization, concurrency control, and recovery, that affect the performance of query processors significantly. We also identified techniques for efficient implementation of our architecture that have no counterpart in non-temporal query processors. Finally, we implemented a prototype query processor for

the temporal query language TQuel, in which views are updated incrementally, to show that our architecture is sufficient to process standard historical queries in TQuel incrementally. The extensive applicability of techniques for efficient implementation of query processors in RDBMS's to our architecture and the results of the prototyping show that our extension of the snapshot algebra can serve as the underlying evaluation mechanism for historical query processing in TDBMS's.

In summary, the results of our research show that the snapshot algebra can be extended to support the incremental update of materialized views in temporal databases to account for updates to their underlying relations.

1.4 Scope of Research

Languages for database query and update exist at no less than three levels of database abstraction. At the user-interface level, calculus-based languages such as SQL are available for expressing query and update operations. At the algebraic level, the snapshot algebra is the formal, abstract language for expressing these same operations. Finally, at the physical level, query and update operations can be defined in terms of data structures and access strategies.

To bound the scope of this research, we restricted our research to language definition at the algebraic level only; we didn't consider language definition at either the user-interface or physical level. Also, we restricted our research to the relational data model; we didn't consider the addition of time to other data models (e.g., network, hierarchical) or to non-relational DBMS's. We addressed only the problem of extending the snapshot algebra to support valid time and transaction time. Hence, we studied the addition of time to only one component of the relational data model. We did not consider the addition of time to other components of the model. For example, we didn't consider temporal keys, functional dependencies, or integrity constraints, because these issues, although important to a temporal extension of the relational data model, are separate from the algebra itself. We also didn't consider the many other issues that arise when one attempts to extend a RDBMS to support time directly. For example, we did not consider temporal query languages or the physical storage of temporal relations. Several studies on each of these issues have already been published. Finally, we restricted our prototype query processor for TQuel to the standard TQuel historical query without aggregates. Also, we didn't require that the prototype be implemented efficiently. Our purpose in implementing the prototype was to show that our architecture is sufficient to process TQuel queries incrementally, not to evaluate the effect of various optimization techniques on the performance of an implementation of our architecture.

1.5 Structure of the Dissertation

In this section we review the organization of the dissertation itself. We describe briefly the contents of each chapter.

This chapter describes the motivation, problem, approach, and scope of this research. Also, basic terminology used in the dissertation is introduced. Chapter 2 reviews related work in defining languages for query and update of temporal databases at the user-oriented and physical levels.

Chapter 3 defines a historical algebra. Formal definitions are provided for a historical relation, ten algebraic operators, and two historical aggregate functions.

Chapter 4 defines a technique for extending the snapshot algebra and our historical algebra to support both aspects of transaction time, evolution of a database's scheme and evolution of a database's contents. A language, whose primary constructs are commands and expressions, is defined to handle transaction time. Commands specify changes to a database while expressions occur within commands and denote a single snapshot or historical state. Expressions are restricted to allowable expressions in the snapshot algebra or our historical algebra, extended to include two new operators that support rollback operations. The extensions are formalized using denotational semantics.

Chapter 5 shows that the algebraic language for query and update of temporal databases defined in Chapters 3 and 4 has the expressive power of the temporal query language TQuel. For each type of TQuel statement (i.e., `retrieve`, `create`, `append`, `replace`, `delete`, and `destroy`), an equivalent algebraic expression is presented. Also, algebraic expressions corresponding to retrieve statements containing aggregates, as well as the basic retrieve statement without aggregates, are presented.

Chapter 6 extends the language defined in Chapter 4 to accommodate views and a spectrum of view maintenance strategies. To support incremental view materialization, incremental versions of both the snapshot algebra and the historical algebra, introduced in Chapter 3, are defined. Operators are redefined as operations on sets of changes to relations rather than as operations on relations themselves. These incremental versions of the algebras are essential to the techniques for incremental view materialization presented in Chapter 7. The incremental versions of the algebras are defined using techniques for incremental evaluation of expressions in the snapshot algebra [Blakeley et al. 1986A, Hanson 1987A, Horwitz 1986].

Chapter 7 describes an architecture for query processing in TDBMS's that accommodates incremental maintenance of materialized historical views. In this architecture, historical queries are represented as update networks in which the internal nodes implement incremental historical operators as defined in Chapter 6. Implementation issues, including query optimization, processing strategies, concurrency control, and recovery, are analyzed as they relate to the architecture. Also, a prototype incremental query processor for TQuel, which shows that the architecture is sufficient to process standard historical queries in TQuel incrementally, is described.

Chapter 8 is an evaluation of algebraic languages for query and update of temporal databases. Ten proposals for extending the snapshot algebra to support either valid time or transaction time are described in terms of the types of objects they support and the operations on object instances they allow. Also, 29 criteria for evaluating these algebraic

languages are presented. Incompatibilities among the criteria are identified and a maximal subset of criteria is defined. The 10 proposals for extending the snapshot algebra to handle one or more aspects of time, along with the language defined in the earlier chapters, are evaluated against the criteria; the language defined here comes closest to satisfying the maximal subset of criteria.

Chapter 9 presents conclusions and discusses future work.

1.6 Notational Conventions

Throughout the dissertation, a **fixed-width** font is used for elements of syntactic categories and a **SMALLCAPS** font is used for elements of semantic domains. Semantic functions appear in **boldface**; all other functions appear in *Italics* with at least the first letter capitalized. Variables in mathematical expressions appear in lower-case *italics*. Appendix A describes the symbols used in the paper and identifies the page where each symbol is either defined or first used. An index to the definitions of terms appears at the end of the paper.

Chapter 2

Previous Work

In this chapter we review briefly previous work relevant to the problem of adding time to the relational data model and RDBMS's. First, we consider efforts to add an aspect of time to RDBMS's, reviewing temporal query languages and extensions of the snapshot algebra that support one or more aspects of time. Then, we consider efforts to resolve implementation issues for TDBMS's, reviewing strategies for storing and accessing temporal relations and strategies for efficient query processing in TDBMS's. Because of the lack of research in the later area, we also review strategies for efficient query processing in RDBMS's that are applicable to TDBMS's.

2.1 Temporal Query Languages

Several temporal query languages have been defined. Clifford proposed that the intensional logic *IL_s*, a typed, higher order lambda calculus with indexical semantics, be used to express queries on historical databases [Clifford & Warren 1983]. The other temporal query languages that have been proposed are derivatives of either Quel [Held et al. 1975], the calculus-based query language for the INGRES relational database management system [Stonebraker et al. 1976], or SQL, the query language for the System R database system [IBM 1981]. TQuel [Snodgrass 1987], HQUEL [Tansel & Arkun 1986], and HTQUEL [Gadia & Vaishnav 1985] are all extensions of Quel. Ben-Zvi's query language for his Time Relational Model (TRM) [Ben-Zvi 1982], TOSQL [Ariav 1984, Ariav 1986], and TSQL [Navathe & Ahmed 1987] are all derivatives of SQL. TQuel, TRM, and TOSQL support both valid time and transaction time. The other languages support only valid time.

Although these temporal query languages have different constructs, most include new constructs for specifying the same basic types of temporal operations. For example, most of these languages provide a new construct, which we term *temporal selection*, to specify a selection predicate for tuples that participate in a query based on their valid times. Also, most provide a new construct, which we term *temporal projection*, to specify the valid times of output tuples as a function of the valid times of their underlying tuples. Finally, most provide temporal versions of the standard aggregates and some provide new

temporal aggregates. To illustrate the types of temporal constructs found in temporal query languages, we now review the language TQuel, emphasizing the new temporal constructs it adds to the basic Quel constructs.

TQuel (*Te_nporal QUEry Language*) [Snodgrass 1987] is an extension of Quel that handles both valid time and transaction time. TQuel is the only query language that supports all four types of relations, snapshot, rollback, historical, and temporal. Also, because it is a superset of Quel, all legal Quel statements are valid TQuel statements. The semantics of TQuel has been defined using tuple relational calculus.

Three new syntactic and semantic constructs are provided to support time. The **valid** clause is the temporal analogue to Quel's target list. It contains two temporal expressions, each consisting of tuple variables, temporal constants, and the temporal constructors **begin of**, **end of**, **overlap**, and **extend**. These two expressions specify the end-points of the interval of validity for output tuples in the derived relation. The **when** clause is the temporal analogue to Quel's where clause. It contains a temporal predicate consisting of temporal expressions, the temporal predicate operators **precede**, **overlap**, and **equal**, and the logical operators **or**, **and**, and **not**. (Note that **overlap** is overloaded; it may be either a temporal constructor or a temporal predicate operator, with context differentiating the uses.) This temporal predicate specifies the temporal selection criteria for input tuples. A third new construct, the **as of** clause, is provided to handle transaction time. It contains either one or two temporal expressions that specify the transaction time(s) for rolling back a rollback or temporal relation in time. The **retrieve** statement is augmented with the **valid**, **when**, and **as of** clauses while the **append**, **delete**, and **replace** statements are augmented with the **valid** and **when** clauses. The **create** command is extended to specify the type of relation being created.

A rich set of aggregates is also defined in TQuel. TQuel aggregates [Snodgrass et al. 1987] are a superset of the Quel aggregates. Hence, each of Quel's six non-unique aggregates (i.e., **count**, **any**, **sum**, **avg**, **min**, and **max**) and three unique aggregates (i.e., **countU**, **sumU**, and **avgU**) has a TQuel counterpart. The TQuel version of each of these aggregates performs the same fundamental operation as its Quel counterpart, with one significant difference. Because a historical relation represents the changing value of its attributes and aggregates are computed from the entire relation, aggregates in TQuel return a distribution of values over time. Hence, while in Quel an aggregate with no by-list returns a single value, in TQuel the same aggregate returns a set of values, each assigned its valid times. When there is a by-list, an aggregate in TQuel returns a distribution of aggregate values over time for each value of the attributes in the by-list. There are also several other TQuel aggregates that do not have Quel counterparts: standard deviation (**stdev** and **stdevU**), average time increment (**avgti**), the variability of time spacing (**varts**), oldest value (**first**), newest value (**last**), interval of validity with the earliest left-most end-point (**earliest**), and interval of validity with the latest right-most end-point (**latest**). All TQuel aggregates have been defined using tuple relational calculus.

Five qualifying clauses may be specified in a TQuel aggregate: the **by** and **where** clauses allowed in Quel aggregates, **when** and **as of** clauses, and a new **for** clause, found

only in aggregates. The *for* clause is used to specify an *aggregation window function* that, along with the time granularity being used, determines an aggregation window for each time t . Only tuples whose interval of validity overlaps the aggregation window for time t participate in the computation of the aggregate's value at time t . Key words (e.g., *each instant, ever, each day, each year*) are used in the *for* clause to define the length of aggregation windows.

2.2 Algebras

Extending the snapshot algebra to include an aspect of time is another topic that has received considerable research interest. Over the past decade, 10 algebras, each an extension of the snapshot algebra that supports one or more aspects of time, have been proposed. Algebras have been defined for LEGOL 2.0 [Jones et al. 1979], Ben-Zvi's Time Relational Model [Ben-Zvi 1982], Clifford's Historical Relational Data Model [Clifford & Croker 1987], Gadia's homogeneous and multihomogeneous models [Gadia 1986, Gadia 1988], Gadia's and Yeung's heterogeneous models [Gadia & Yeung 1988, Yeung 1986], and Navathe's Temporal Relational Model [Navathe & Ahmed 1986]. Lorentzos, Johnson, Sadeghi, Sarda, and Tansel also have defined algebras [Lorentzos & Johnson 1987A, Sadeghi 1987, Sarda 1988, Tansel 1986]. While all these algebras support valid time, only Ben-Zvi's algebra supports transaction time.

The algebras differ both in the types of objects they define and in the kinds of operations they provide. These differences are the result of choices to several basic design decisions. For example, some of the algebras associate valid time with tuples while others associate valid time with attributes. Also, some of the algebras retain the set-theoretic semantics of the basic relational operators and introduce new operators to deal with the temporal dimension of data while others extend the semantics of the relational operators to deal with the temporal dimension of data directly. We provide a review of all 10 algebras in Chapter 8.

2.3 Storage Structures and Access Strategies

While there has been considerable research into an appropriate query language and algebra for a TDBMS based on a temporal extension of the relational model, there has been only limited research into the implementation of such a TDBMS. One implementation issue, however, that has received some research attention is the appropriate storage structures and access strategies for temporal databases. Information, once added to a historical relation, can be deleted, but only to correct errors. Information, once added to a rollback or temporal relation, can never be deleted; otherwise, the rollback operation could not be supported. Because of these properties, the volume of information that must be maintained for a historical, rollback, or temporal version of a relation will be substantially greater than that for a corresponding snapshot version of the relation. Hence, appropriate storage structures

and access strategies are even more important implementation issues for TDBMS's than for RDBMS's.

Ahn has proposed both storage structures and access strategies for temporal relations [Ahn 1986A]. He proposed a *temporally partitioned store* for temporal relations in which the *current store* holds the current version of all tuples in the relation and the *history store* holds the past versions of all tuples. This temporally partitioned store allows different storage formats and different storage media to be used for the different stores. Ahn investigated the relative advantages and disadvantages of several different storage formats for the history store, including reverse chaining, indexing, clustering, stacking, and cellular chaining. He also introduced a new hashing technique, termed *nonlinear hashing*, to cluster the past versions of a tuple in the history store. Finally, Ahn implemented a prototype of a TDBMS to study the performance of various storage structures for temporal databases. He showed the feasibility of adding a temporal dimension to RDBMS's without incurring a performance penalty for conventional non-temporal queries. He also showed that specific storage structures can be used to improve the performance of various types of temporal queries.

Thirumalai and Krishna have proposed a three-level, rather than a two-level, storage structure for temporal relations [Thirumalai & Krishna 1988]. In their proposal, a *current store* holds the tuples that are currently both active (i.e., have yet to be logically deleted) and valid (i.e., valid time overlaps the present), a *history store* holds the tuples that are active, but no longer valid, and a *relic store* holds the tuples that are both inactive and no longer valid. Only the current store is needed to answer non-temporal queries, while the history store is needed to answer historical queries and the relic store is needed to answer rollback queries. They investigated organization of the history store as a grid file [Nievergelt et al. 1984] to cluster tuples by both key and valid time.

Segev and Shoshani captured the semantics of valid time in historical relations through the concept of *time sequences* [Segev & Shoshani 1987]. A time sequence is an ordered sequence in the time domain of values for a database entity (e.g., someone's employment history). Rotem and Segev proposed multidimensional file partitioning as an appropriate storage structure for time sequences [Rotem & Segev 1987]. They studied two alternatives for multidimensional partitioning, termed symmetric and asymmetric partitioning, and showed through simulation experiments that asymmetric partitioning has a performance advantage in terms of disk accesses.

The storage structure for POSTGRES [Stonebraker 1987] supports rollback relations using a temporally partitioned store similar to that of Ahn. The current version of each tuple is stored on magnetic disk while past versions of tuples are asynchronously moved to an archival medium, perhaps a write-once-read-many (WORM) optical disk. An arbitrary number of secondary indexes can be specified for the archived portion of a rollback relation to support temporal queries and the indexes need not be the same as those for the magnetic disk portion of the relation. Performance studies have indicated that the POSTGRES storage structure is competitive with conventional storage structures.

Lum, et al. proposed a data structure for rollback relations in which the current

version of tuples and the past versions of tuples are all maintained on-line, but in separate relations [Lum et al. 1984]. Stored with the current version of each tuple are a time-stamp and a pointer to a linked list of the tuple's past versions. Past versions of the tuple are stored in a separate relation, linked in reverse time order. To allow random access to tuples and their histories, a *current index tree* is maintained for the current store and an *history index tree* is maintained for the history store. These trees are conventional structures, such as B-trees or B*-trees whose leaves are (index value, pointer) pairs.

The considerable research in appropriate data structures and access strategies for persistent data structures also has application to temporal databases. Persistent data structures, like rollback and temporal relations, maintain a record of their evolution over time resulting from the execution of insert and delete operations. Dobkin and Munro proposed a persistent data structure for ordered lists [Dobkin & Munro 1980, Dobkin & Munro 1985]. Their data structure records the evolution of an ordered list over time by remembering the rank history of each list element. Queries concerning an element's rank can be posed as of "now" or some time in the past. Overmars proposed methods for handling the persistent list problem that improve on the algorithms of Dobkin and Munro [Overmars 1981A, Overmars 1981B, Overmars 1983]. Chazelle and Cole both proposed persistent data structures for a sorted set; Chazelle using *canal* trees and Cole using binary search trees to record a set's evolution [Chazelle 1985, Cole 1986]. Queries concerning set membership or an element's neighbors can be posed as of "now" or some time in the past. Myers proposed a persistent data structure for both sorted sets and lists, either ordered or unordered [Myers 1984]. His approach, which is called *path copying* elsewhere [Sarnak & Tarjan 1986], is based on the representation of a set or list at time t as a height-balanced tree (e.g., AVL tree). On update, nodes on the path from the tree's root to the point of update are copied and then linked to all subtrees not on the path to form a new tree. Sarnak and Tarjan propose a variation of path copying that requires an amortized space cost of only $\theta(1)$ per update [Sarnak & Tarjan 1986].

2.4 Strategies for Efficient Query Processing

Strategies for efficient query processing in TDBMS's is an open research topic. Gadia provided a computational semantics for his historical algebra to support its efficient implementation [Gadia 1988] and Tansel provided algebraic tautologies for his new temporal operators that can be used in query optimization [Tansel 1986]. Otherwise, implementation issues related to the use of a historical algebra as the evaluation mechanism for queries in a TDBMS have yet to be explored.

Although there has been a lack of research in strategies for efficient query processing in TDBMS's, many of the strategies for efficient query processing in RDBMS's are likely to have an analogue in TDBMS's. For example, much of the substantial research in the optimization of single snapshot algebra expressions [Aho et al. 1979, Ceri & Gottlob 1985, Freytag & Goodman 1986, Hall 1976, Selinger et al. 1979, Smith & Chang 1975, Ullman 1982, Wong & Youssef 1976] and multiple snapshot algebra expressions [Finkelstein 1982,

Roussopoulos 1982A, Satoh et al. 1985, Sellis & Shapiro 1985] is likely to have an analogue for historical algebras. Also, recently developed strategies for maintaining materialized database views appear directly applicable to the processing of recurring historical queries in TDBMS's.

Derived relations, which are defined as algebraic expressions involving other relations, are either unnamed or named [Date 1986B]. Unnamed derived relations are simply the results of queries while named derived relations may be classified as either *views* [Date 1986C] or *snapshots* [Adiba & Lindsay 1980]. Views and snapshots differ in that, from a user's perspective, views change over time to reflect changes in their underlying relations, whereas snapshots, once evaluated, are unaffected by subsequent changes in their underlying relations. Traditionally, query modification has been used to convert queries against a view into queries against the view's underlying relations [Stonebraker 1975]. Recently, however, research has focused on strategies for maintaining materialized views, where the views are incrementally updated to reflect changes in the views' underlying relations [Blakeley et al. 1986A, Horwitz & Teitelbaum 1986, Roussopoulos & Kang 1986A, Roussopoulos & Kang 1986B, Shmueli & Atai 1984, Shmueli & Itai 1987]. Sufficient and necessary conditions for detecting updates of base relations that cannot affect views have been identified [Blakeley et al. 1986B] and an incremental version of the snapshot algebra has been defined [Blakeley et al. 1986A]. Also, several architectures for incremental view materialization have been proposed [Horwitz 1985, Roussopoulos 1982A, Roussopoulos 1982B, Snodgrass 1982]. Hanson showed that, for at least some classes of queries against views, incremental view materialization strategies have performance advantages over query modification strategies [Hanson 1987A]. Also, recurring queries can be implemented as materialized views to reduce the amortized cost of their evaluations. Roussopoulos showed that incremental view materialization is more efficient than query re-evaluation as an implementation strategy for many types of recurring queries [Roussopoulos 1987].

Chapter 3

Supporting Valid Time: A Historical Algebra

As discussed in Chapter 1, there are three orthogonal aspects of time that a DBMS should support: valid time, transaction time, and user-defined time. Although the snapshot algebra [Codd 1970] supports user-defined time, it supports neither valid time nor transaction time. In this chapter we extend the snapshot algebra to handle valid time by defining a *historical algebra*. We do not consider here any extension, of either the snapshot algebra or our historical algebra, to support transaction time. In the next chapter we describe an approach for adding transaction time to both the snapshot algebra and our historical algebra. This approach also applies without change to most other historical algebras supporting valid time. Because valid time and transaction time are orthogonal, they can be studied in isolation.

Several benefits accrue from defining a historical algebra that extends the snapshot algebra to support valid time. A historical algebra is essential to the formulation of a historical data model because it defines formally the types of objects and the operations on object instances allowed in the data model. The usefulness of a historical data model in representing the time-varying aspect of real-world phenomena depends on the power and expressiveness of its underlying historical algebra. Similarly, the algebra determines a data model's support of calculus-based query languages. Also, implementation issues, such as query optimization and physical storage strategies, can best be addressed in terms of the algebra.

3.1 Approach

The snapshot algebra allows us to model reality only at a single time. We want to extend the snapshot algebra to model reality over an interval rather than at a single time. To do so, we redefine a *relation*, the only type of object allowed in the algebra, to include valid time. We also redefine the algebraic operators, and introduce new operators, to handle this new temporal dimension.

To extend objects in the snapshot algebra to include valid time, we had to make three basic design decisions.

- Is valid time associated with tuples (as additional implicit attributes) or with attributes?
- How is valid time represented? Do time-stamps, which represent valid time, correspond to chronons, intervals, or sets of chronons, not all of which are consecutive?
- Are attributes required to be atomic-valued or are they allowed to be set-valued? If set-valued attributes are allowed, then the first-normal-form property of the snapshot algebra cannot be satisfied [Codd 1970].

We chose to

- Associate valid time with attributes rather than with tuples,
- Represent valid time as a set of (not necessarily consecutive) chronons, and
- Require that the value component of attributes be atomic-valued but allow the valid-time component of attributes to be set-valued.

To extend operations in the snapshot algebra to handle valid time, we had to make two subsequent design decisions.

- Is the set-theoretic semantics of the basic relational operators retained and new operators introduced to deal with the temporal dimension of the real-world phenomena being modeled or is the semantics of the relational operators extended to account for the temporal dimension directly? If the latter, then how do these operators compute the valid time of attributes in resulting tuples?
- How does the algebra handle *temporal selection* (i.e., tuple selection based on valid times), *temporal projection* (i.e., computation of new valid times for a tuple's attributes from their current valid times), and *temporal aggregation* (i.e., computation of a distribution of aggregate values over time); operations that are unique to a historical algebra?

We chose to

- Extend the semantics of the relational operators to account for the temporal dimension directly and redefine the operators formally to specify how each computes the valid time of attributes in resulting tuples, and
- Introduce new operators to handle temporal selection, projection, and aggregation.

Our choices for these five design decisions reflect our goal to define a historical algebra that has as many of the most desirable properties of a historical algebra as possible. For example, we wanted the historical algebra to be a straightforward extension of the snapshot algebra so that relations and algebraic expressions in the snapshot algebra would have equivalent counterparts in the historical algebra. Yet we also wanted the algebra to support historical queries and adhere to the user-oriented model of historical relations as three-dimensional objects, where the additional, third dimension is valid time [Ariav 1986, Ariav & Clifford 1986, Clifford & Tansel 1985]. Hence, we did not restrict historical relations to first-normal-form (i.e., we allow set-valued time-stamps), insist on time-stamping of entire tuples, or require that time-stamps be atomic-valued because each of those restrictions would have prevented the algebra from having other, more highly desirable properties. All design decisions were made so that the resulting algebra would possess a maximal set of desirable properties. In Chapter 8 we present a detailed discussion of desirable properties of historical algebras as well as an evaluation of our algebra and the historical algebras proposed by others, using the identified properties as evaluation criteria. We also review our design decisions, considering these evaluation criteria.

Efficient direct implementation of the algebra was not one of our primary design objectives. Rather, our goal was to define an algebra that preserves the associative, commutative, and distributive properties of the snapshot algebra in order that optimization strategies developed for the snapshot algebra can be applied in implementations of the historical algebra. Our formulation of the algebraic operators would be inefficient if mapped directly into an implementation. While we can envision more efficient implementations, incorporating such efficiencies in the semantics would have made it much more complex. Finally, we expect that new optimization strategies, unique to the historical algebra, also will be used in its implementation. We discuss these issues further in Chapter 7.

In the following sections we define our historical algebra, presenting formal definitions for a historical relation, six algebraic operators, and two historical aggregate functions. We then show that all the operators preserve the value-equivalence property of historical relation states, which we define in the next section. Finally, we conclude this chapter by discussing briefly techniques that can be used to extend the algebra defined here to handle periodicity, multi-dimensional time-stamps, and non-first-normal-form historical relations.

3.2 Historical Relation

We define a *historical relation* in terms of its *scheme* and the set of *states* that it may assume. The relation's structure is defined by the scheme; its contents may be any one of the allowable states.

As we saw in Chapter 1, the definition of a relation's scheme in terms of the relation's class and attributes is sufficient, even if we allow databases to contain relations of all four classes. Assume that we are given an arbitrary set of syntactic identifiers *IDENTIFIER* and the e arbitrary, non-empty, finite or denumerable sets \mathcal{D}_u , $1 \leq u \leq e$. Let z be a

function that maps each identifier I in the set $IDENTIFIER$ onto one of the sets \mathcal{D}_u , $1 \leq u \leq e$, or the special element UNBOUND.

$$z : IDENTIFIER \rightarrow \{\mathcal{D}_1, \dots, \mathcal{D}_e, \text{UNBOUND}\}$$

If z maps an identifier I onto a set \mathcal{D}_u , we refer to I as an *attribute name*, or simply an *attribute* and \mathcal{D}_u as its *value domain*. Hence, the function z , which we refer to hereafter as the relation *signature*, induces the set of attributes $\mathcal{A} = \{I \mid z(I) \neq \text{UNBOUND}\}$. A historical relation's scheme is then simply the relation class HISTORICAL and a relation signature z .

We now define the set of states that a historical relation may assume, given a relation signature z . Let T be the set of positive integers, where each element of T represents a chronon. Assume that, if t_1 immediately precedes t_2 in the linear ordering of T , then t_1 represents the interval $[Startof(t_1), Startof(t_2))$, where $Startof$ is a function that maps a chronon in the discrete model onto the "point" in the continuous model that corresponds to the chronon's beginning. The granularity of time (e.g., nanosecond, month, year) associated with T is arbitrary. Let $\wp(T)$ be the power set of T . An element of $\wp(T)$ is then a set of integers, each of which represents a chronon. Also, any group of consecutive integers t_1, \dots, t_n appearing in an element of $\wp(T)$, together represent the interval $[t_1, t_n + 1)$. If we let $\wp(T)$ be the *time domain* for each attribute in \mathcal{A} , we can define a *historical tuple* ht as a function that maps each attribute in \mathcal{A} onto an ordered pair from the attribute's value and time domains.

$$ht : \mathcal{A} \rightarrow (\mathcal{D}_1 + \dots + \mathcal{D}_e, \wp(T))$$

with the following restrictions:

- $\forall I, I \in \mathcal{A}, Value(ht(I)) \in z(I)$ and
- $\exists I, I \in \mathcal{A}, Valid(ht(I)) \neq \emptyset$.

Here, the notation "+" on domains means the disjoint union of domains, the function *Value* maps an attribute onto its value component, and the function *Valid* maps an attribute onto its valid-time component. (Formal definitions for both appear in Appendix B.)

Note that it is possible for all but one of a tuple's attributes to have an empty time-stamp. If an attribute's time-stamp is empty, then its valid time is assumed to be unknown. Hence, empty attribute time-stamps can be thought of as corresponding to temporal nulls. We allow tuples to contain temporal nulls for some, but not all, attributes.

We define two tuples, ht and ht' , to be *value-equivalent* if and only if $\forall I, I \in \mathcal{A}, Value(ht(I)) = Value(ht'(I))$. A *historical state* is then defined as a finite set of historical tuples, with the restriction that no two tuples in the state are value-equivalent. \mathcal{H} , represents the domain of all historical states, consistent with the relation signature z , that a historical relation may assume.

EXAMPLE. Assume that we are given the relation signature *Student* with attributes {*sname*, *course*} and the following set of tuples over this relation signature. For this and

all later examples, assume that the granularity of time is a semester relative to the Fall semester 1980. Hence, 1 represents the Fall semester 1980, 2 represents the Spring semester 1981, etc.

$$S = \{ \langle ("Phil", \{1, 3\}), ("English", \{1, 3\}) \rangle, \\ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \\ \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}) \rangle, \\ \langle ("Phil", \{4\}), ("English", \{4\}) \rangle \}$$

For notational convenience we enclose each attribute value in parentheses and each tuple in angular brackets (i.e., $\langle \rangle$). Also for notational convenience, we assume the natural mapping between attribute names and attribute values (e.g., `sname` \rightarrow $("Phil", \{1, 3\})$, and `course` \rightarrow $("English", \{1, 3\})$). Note that S is not an allowable historical state because there are value-equivalent tuples in the set (the first and fourth tuples are value-equivalent). If we replace the two value-equivalent tuples in S with a single tuple, then the new set S_1 is a historical state in $\mathcal{H}_{Student}$.

$$S_1 = \{ \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}) \rangle, \\ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \\ \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}) \rangle \} \quad \square$$

In summary, the historical algebra places the same basic restrictions on the value components of attributes as the snapshot algebra places on attribute values. Neither set-valued attribute value components nor tuples with duplicate attribute value components are allowed. Valid time, however, is represented by a set-valued time-stamp that is associated with individual attributes. A time-stamp represents possibly disjoint intervals and the time-stamps assigned to two attributes in a given tuple may, but need not, be identical.

3.3 Historical Operators

We present eight operators that serve to define the historical algebra. Five of these operators — union, difference, cartesian product, projection, and selection — are analogous to the five operators that serve to define the snapshot algebra for snapshot states [Ullman 82]. Each of these five operators on historical states is represented as op to distinguish it from its snapshot algebra counterpart op . Historical derivation is a new operator that replaces the time-stamp of each attribute in a tuple with a new time-stamp, where the new time-stamps are computed from the existing time-stamps of the tuple's attributes. The remaining two operators, aggregation and unique aggregation, compute aggregates. After defining the operators, we show that all eight preserve the value-equivalence property of historical states.

EXAMPLE. The three historical states S_1 , S_2 , and S_3 are used in the examples that accompany the definitions of the operators. S_2 , like S_1 , is a historical state over the relation signature *Student* with attributes {*sname*, *course*}. S_3 is a historical state over the relation signature *Home* with attributes {*hname*, *state*}. While the attributes of a tuple in S_1 , S_2 , and S_3 have the same time-stamp, in general, attributes within a tuple can have different time-stamps.

$$S_2 = \{ \langle ("Phil", \{3, 4\}), ("English", \{3, 4\}) \rangle, \\ \langle ("Norman", \{7\}), ("Math", \{7\}) \rangle, \\ \langle ("Tom", \{5, 6\}), ("English", \{5, 6\}) \rangle \}$$

$$S_3 = \{ \langle ("Phil", \{1, 2, 3\}), ("Kansas", \{1, 2, 3\}) \rangle, \\ \langle ("Phil", \{4, 5, 6\}), ("Utah", \{4, 5, 6\}) \rangle, \\ \langle ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\ \langle ("Norman", \{7, 8\}), ("Texas", \{7, 8\}) \rangle \} \quad \square$$

3.3.1 Union

Let Q and R be historical states of m -tuples over the relation signature z with attributes $\mathcal{A} = \{I_1, \dots, I_m\}$. Then $Q \cup R$, the historical union of Q and R , is defined as

$$Q \cup R \triangleq \{ q^m \mid Q(q) \wedge \neg(\exists r, r \in R \wedge \forall I, I \in \mathcal{A}. Value(q(I)) = Value(r(I))) \} \\ \cup \{ r^m \mid R(r) \wedge \neg(\exists q, q \in Q \wedge \forall I, I \in \mathcal{A}. Value(r(I)) = Value(q(I))) \} \\ \cup \{ u^m \mid \exists q \exists r, q \in Q \wedge r \in R \wedge \forall I, I \in \mathcal{A}, \\ Value(u(I)) = Value(q(I)) = Value(r(I)) \\ \wedge Valid(u(I)) = Valid(q(I)) \cup Valid(r(I)) \}$$

$Q \cup R$ is the set of tuples that are in Q , R , or both, with the restriction that each pair of value-equivalent tuples is represented by a single tuple. Note that if a tuple in Q and a tuple in R are value-equivalent, then they are represented in $Q \cup R$ by a single tuple. The time-stamp associated with each attribute of this tuple in $Q \cup R$ is the set union of the time-stamps of the corresponding attribute in the value-equivalent tuples in Q and R .

EXAMPLE. $S_1 \cup S_2 = \{ \langle (\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\}) \rangle, \langle (\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}) \rangle, \langle (\text{"Norman"}, \{5, 6, 7\}), (\text{"Math"}, \{5, 6, 7\}) \rangle, \langle (\text{Tom}, \{5, 6\}), (\text{"English"}, \{5, 6\}) \rangle \}$ □

3.3.2 Difference

Let Q and R be historical states of m -tuples over the relation signature z with attributes $\mathcal{A} = \{I_1, \dots, I_m\}$. Then $Q - R$, the historical difference of Q and R , is defined as

$$\begin{aligned} Q - R &\triangleq \{ q^m \mid Q(q) \wedge \neg(\exists r, r \in R \wedge \forall I, I \in \mathcal{A}, \text{Value}(q(I)) = \text{Value}(r(I))) \} \\ &\cup \{ u^m \mid (\exists q \exists r, q \in Q \wedge r \in R \wedge \forall I, I \in \mathcal{A}, \\ &\quad \text{Value}(u(I)) = \text{Value}(q(I)) = \text{Value}(r(I)) \\ &\quad \wedge \text{Valid}(u(I)) = \text{Valid}(q(I)) - \text{Valid}(r(I))) \\ &\quad \wedge (\exists I, I \in \mathcal{A} \wedge \text{Valid}(u(I)) \neq \emptyset) \} \\ &\} \end{aligned}$$

$Q - R$ is the set of all tuples that satisfy three criteria. First, a tuple in $Q - R$ must have a value-equivalent counterpart in Q . Second, the time-stamp of each attribute of a tuple in $Q - R$ must equal the set difference of the time-stamps of the corresponding attribute in the value-equivalent tuple in Q and the value-equivalent tuple in R , if any. Third, the time-stamp of at least one attribute of each tuple in $Q - R$ must be non-empty.

EXAMPLE. $S_1 - S_2 = \{ \langle (\text{"Phil"}, \{1\}), (\text{"English"}, \{1\}) \rangle, \langle (\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}) \rangle, \langle (\text{"Norman"}, \{5, 6\}), (\text{"Math"}, \{5, 6\}) \rangle \}$ □

3.3.3 Cartesian Product

Let Q be a historical state of m_1 -tuples on the relation signature z_Q with attributes $\mathcal{A}_Q = \{I_{Q,1}, \dots, I_{Q,m_1}\}$ and R be a historical state of m_2 -tuples on the relation signature z_R with attributes $\mathcal{A}_R = \{I_{R,1}, \dots, I_{R,m_2}\}$. Also assume that $\mathcal{A}_Q \cap \mathcal{A}_R = \emptyset$. Then $Q \times R$, the historical cartesian product of Q and R , is defined as

$$\begin{aligned}
Q \hat{\times} R \triangleq & \{ u^{m_1+m_2} \mid (\exists q, q \in Q \wedge \forall I, I \in \mathcal{A}_Q, \text{Value}(u(I)) = \text{Value}(q(I))) \\
& \wedge \text{Valid}(u(I)) = \text{Valid}(q(I))) \\
& \wedge (\exists r, r \in R \wedge \forall I, I \in \mathcal{A}_R, \text{Value}(u(I)) = \text{Value}(r(I))) \\
& \wedge \text{Valid}(u(I)) = \text{Valid}(r(I))) \\
& \}
\end{aligned}$$

The cartesian product operator for historical states is identical to the cartesian product operator for snapshot states. $Q \hat{\times} R$ is the set of $(m_1 + m_2)$ -tuples, each of which is formed from a m_1 -tuple in Q and a m_2 -tuple in R . While our definition of cartesian product requires that the attributes defined by the signatures z_Q and z_R be disjoint, we could eliminate this last restriction and effectively allow the cartesian product of snapshot states on arbitrary signatures through the introduction of a simple attribute renaming operator [Maier 1983].

EXAMPLE.

$$\begin{aligned}
S_1 \hat{\times} S_3 = & \{ ((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\}), (\text{"Phil"}, \{1, 2, 3\}), (\text{"Kansas"}, \{1, 2, 3\})), \\
& ((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\}), (\text{"Phil"}, \{4, 5, 6\}), (\text{"Utah"}, \{4, 5, 6\})), \\
& ((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\}), (\text{"Norman"}, \{1, 2, 5, 6\}), (\text{"Utah"}, \{1, 2, 5, 6\})), \\
& ((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\}), (\text{"Norman"}, \{7, 8\}), (\text{"Texas"}, \{7, 8\})), \\
& ((\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}), (\text{"Phil"}, \{1, 2, 3\}), (\text{"Kansas"}, \{1, 2, 3\})), \\
& ((\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}), (\text{"Phil"}, \{4, 5, 6\}), (\text{"Utah"}, \{4, 5, 6\})), \\
& ((\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}), (\text{"Norman"}, \{1, 2, 5, 6\}), (\text{"Utah"}, \{1, 2, 5, 6\})), \\
& ((\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\}), (\text{"Norman"}, \{7, 8\}), (\text{"Texas"}, \{7, 8\})), \\
& ((\text{"Norman"}, \{5, 6\}), (\text{"Math"}, \{5, 6\}), (\text{"Phil"}, \{1, 2, 3\}), (\text{"Kansas"}, \{1, 2, 3\})), \\
& ((\text{"Norman"}, \{5, 6\}), (\text{"Math"}, \{5, 6\}), (\text{"Phil"}, \{4, 5, 6\}), (\text{"Utah"}, \{4, 5, 6\})), \\
& ((\text{"Norman"}, \{5, 6\}), (\text{"Math"}, \{5, 6\}), (\text{"Norman"}, \{1, 2, 5, 6\}), (\text{"Utah"}, \{1, 2, 5, 6\})), \\
& ((\text{"Norman"}, \{5, 6\}), (\text{"Math"}, \{5, 6\}), (\text{"Norman"}, \{7, 8\}), (\text{"Texas"}, \{7, 8\})) \}
\end{aligned}$$

Let this be historical state S_4 with attributes {sname, course, hname, state}. □

3.3.4 Selection

Let R be a historical state of m -tuples on the relation signature z with attributes $\mathcal{A} = \{I_1, \dots, I_m\}$. Also, let F be a boolean function involving

- Attribute names I_1, \dots, I_m ;
- Constants from the value domains to which z maps the attribute names I_1, \dots, I_m ;
- Relational operators $<, =, >$; and
- Logical operators \wedge, \vee , and \neg

where, to evaluate F for a tuple r , $r \in R$, we substitute the value components of the attributes of r for all occurrences of their corresponding attribute names in F . Then the historical selection of R , denoted by $\hat{\sigma}_F(R)$, is defined as

$$\hat{\sigma}_F(R) \triangleq \{r^m \mid r \in R \wedge F(r)\}$$

Thus, $\hat{\sigma}_F(R)$ is simply the set of tuples in R for which F is true.

EXAMPLE.

$$\begin{aligned} \hat{\sigma}_{\text{sname}=\text{hname}}(S_4) = \\ \{ & \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}), ("Phil", \{1, 2, 3\}), ("Kansas", \{1, 2, 3\}) \rangle, \\ & \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}), ("Phil", \{4, 5, 6\}), ("Utah", \{4, 5, 6\}) \rangle, \\ & \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}), ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\ & \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}), ("Norman", \{7, 8\}), ("Texas", \{7, 8\}) \rangle, \\ & \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}), ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\ & \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}), ("Norman", \{7, 8\}), ("Texas", \{7, 8\}) \rangle \} \end{aligned}$$

Let this be historical state S_5 with attributes `{sname, course, hname, state}`. □

3.3.5 Projection

Let R be a historical state of m -tuples on the relation signature z with attributes $\mathcal{A}_R = \{I_{R,1}, \dots, I_{R,m}\}$. Also, assume that we are given a set of identifiers X of cardinality n , where $X \subseteq \mathcal{A}$. Then $\hat{\pi}_X(R)$, the historical projection of R , is defined as

$$\begin{aligned}
\hat{\pi}_X(R) \triangleq & \{ u^n \mid (\forall I, I \in X, \forall t, t \in \text{Valid}(u(I)), \\
& \exists r, (r \in R \\
& \quad \wedge \forall I', I' \in X, \text{Value}(u(I')) = \text{Value}(r(I'))) \\
& \quad \wedge t \in \text{Valid}(r(I))) \\
&) \\
& \wedge (\forall r, (r \in R \wedge \forall I, I \in X, \text{Value}(r(I)) = \text{Value}(u(I))), \\
& \quad \forall I, I \in X, \text{Valid}(r(I)) \subseteq \text{Valid}(u(I)) \\
&) \\
& \wedge (\exists I, I \in X \wedge \text{Valid}(u(I)) \neq \emptyset) \\
\}
\end{aligned}$$

Like the projection operator for snapshot states, the projection operator for historical states retains, for each tuple, only the tuple components that correspond to the attribute names in X . All other tuple components are removed. Value-equivalent tuples in the resulting set are then combined and tuples that have an empty valid component for all tuple components are removed.

$$\begin{aligned}
\text{EXAMPLE. } \hat{\pi}_{\{\text{sname}, \text{state}\}}(S_5) = & \{ \langle ("Phil", \{1, 3, 4\}), ("Kansas", \{1, 2, 3\}) \rangle, \\
& \langle ("Phil", \{1, 3, 4\}), ("Utah", \{4, 5, 6\}) \rangle, \\
& \langle ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\
& \langle ("Norman", \{1, 2, 5, 6\}), ("Texas", \{7, 8\}) \rangle \}
\end{aligned}$$

Let this be historical state S_6 over the relation signature *Enrollment* with the attributes $\{\text{sname}, \text{state}\}$. Also assume that in this historical state the valid-time component of attribute **sname** represents the interval(s) when the specified student was enrolled and that the valid-time component of attribute **state** represents the interval(s) when the student was a resident of the specified state. \square

The operator $\hat{\pi}$ also supports projections on expressions. Rather than simply project a tuple onto a subset of its attributes, the operator may project a tuple onto an arbitrary number of new attributes. Then, the value (or valid-time) component of each new attribute is a function of the value (valid-time) components of the tuple's attributes. Assume that we are given the n arbitrary, but distinct, identifiers I_1, \dots, I_n . Let $Evaluate_l$, $1 \leq l \leq n$, be an arbitrary expression involving the attribute names $I_{R,a}$, $1 \leq a \leq m$, where $Evaluate_l$ is evaluated, for a tuple r , $r \in R$, by substituting the value components of the attributes of r for all occurrences of their corresponding attribute names in $Evaluate_l$. Also, let $Evalid_l$, $1 \leq l \leq n$, be an arbitrary expression involving the attribute names $I_{R,a}$, $1 \leq a \leq m$, where

$Evalid_l$ is evaluated for a tuple r , $r \in R$, by substituting the valid-time components of the attributes of r for all occurrences of their corresponding attribute names in $Evalid_l$. In addition, assume that evaluation of $Evalue_l$ for every tuple r produces an element of the domain D_c , $1 \leq c \leq e$, and that evaluation of $Evalid_l$ produces an element of the domain $\wp(\mathcal{T})$. Then, a version of the projection operator $\hat{\pi}$, more general than that given above, is defined as

$$\begin{aligned} \hat{\pi}_{\{(I_1, (Evalue_1, Evalid_1)), \dots, (I_n, (Evalue_n, Evalid_n))\}}(R) &\triangleq \\ \{u^n \mid & (\forall l, 1 \leq l \leq n, \forall t, t \in Valid(u(I_l))), \\ \exists r, (r \in R & \wedge \forall h, 1 \leq h \leq n, Value(u(I_h)) = Evalue_h(r) \\ & \wedge t \in Evalid_l(r)) \\) & \wedge (\forall r, (r \in R \wedge \forall l, 1 \leq l \leq n, Evalue_l(r) = Value(u(I_l))), \\ \forall h, 1 \leq h \leq n, Evalid_h(r) & \subseteq Valid(u(I_h))) \\) & \wedge (\exists l, 1 \leq l \leq n \wedge Valid(u(I_l)) \neq \emptyset) \\ \} & \end{aligned}$$

Here, the result is a historical state with attributes $\{I_1, \dots, I_m\}$.

EXAMPLE.

$$\begin{aligned} \hat{\pi}_{\{(name, (sname, sname)) (state, (state, sname \cap state))\}}(S_6) &= \\ \{ & ((("Phil", \{1, 3, 4\}), ("Kansas", \{1, 3\})), \\ & ((("Phil", \{1, 3, 4\}), ("Utah", \{4\})), \\ & ((("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\})), \\ & ((("Norman", \{1, 2, 5, 6\}), ("Texas", \{\})) \} \} \} \} \end{aligned}$$

The result is a historical state with attributes $\{name, state\}$ rather than $\{sname, state\}$. The valid-time component of attribute **name** represents the interval(s) when the specified student was enrolled, but the valid-time component of attribute **state** represents only the subinterval(s) of enrollment when the student was a resident of the specified state. Note that, because Norman's enrollment never overlapped his residency in Texas, the valid-time component of the attribute **state** of the fourth tuple is the empty set. \square

3.3.6 Historical Derivation

The historical derivation operator δ is a new operator that does not have an analogous snapshot operator. δ is effectively a combination of temporal selection and projection on a tuple's attribute time-stamps.

Let R be a historical state of m -tuples on the relation signature z with attributes $A = \{I_1, \dots, I_m\}$. For a tuple $r, r \in R$, δ calculates a new valid-time component for each of r 's attributes as a function of selective intervals in r 's attribute time-stamps. The new valid-time component for attribute I_a , $1 \leq a \leq m$, is specified by a temporal function V_a . To compute a new valid-time component for I_a , δ first determines the non-overlapping intervals in each of r 's attribute time-stamps. Then, δ determines all assignments of those intervals to their attribute names for which a boolean function G is true. For each assignment of intervals to attribute names for which G is true, the operator evaluates V_a . The sets of times resulting from the evaluations of V_a are then combined to form a new valid-time component for I_a . The operator has the following form.

$$\delta_{G, \{(I_1, V_1), \dots, (I_m, V_m)\}}(R)$$

EXAMPLE.

$$\delta_{(\text{surname} \cap \text{state}) = \text{name}, \{(\text{surname}, \text{surname}), (\text{state}, \text{surname})\}}(S_6)$$

In this example, the predicate requires that an interval from the valid-time component of attribute `sname` be contained in an interval from the valid-time component of attribute `state`. The new valid-time component of each attribute is simply the union of intervals from `sname`'s time-stamp that satisfy the predicate. We discuss this example further, once we have defined the historical derivation operator formally. \square

Several functions, defined on the domains T and $\mathcal{P}(T)$, are used either directly or indirectly in the definition of the historical derivation operator. Before defining the derivation operator itself, we describe informally these auxiliary functions. Formal definitions appear in Appendix B.

First takes a set of times from the domain $\mathcal{P}(T)$ and maps it onto the earliest time in the set.

Last takes a set of times from the domain $\mathcal{P}(T)$ and maps it onto the latest time in the set.

Pred is the predecessor function on the domain T . It maps a time onto its immediate predecessor in the linear ordering of all times.

Succ is the successor function on the domain \mathcal{T} . It maps a time onto its immediate successor in the linear ordering of all times.

Extend maps two times onto the set of times that represents the interval between the first time and the second time.

Interval maps a set of times onto the set of intervals containing the minimum number of non-disjoint intervals represented by the input set. Each time in the input set appears in exactly one interval in the output set and each interval in the output set is itself represented by a set of times.

EXAMPLE. Consider the following tuple taken from the historical state S_6 defined previously:

$$r = \langle ("Norman", \{1, 2, 5, 6\}), ("Texas", \{7, 8\}) \rangle$$

then

$$\text{Interval}(\text{Valid}(r(\text{sname}))) = \{\{1, 2\}, \{5, 6\}\}$$

$$\text{Interval}(\text{Valid}(r(\text{state}))) = \{\{7, 8\}\}$$

□

Given these auxiliary functions, we can now define the historical derivation operator on historical states. Let V_a , $1 \leq a \leq m$, be a temporal function involving

- Attribute names I_1, \dots, I_m ;
- Constants from the domain \mathcal{IN} of non-disjoint intervals defined in Appendix B;
- Functions *First*, *Last*, and *Extend*; and
- Set operators \cup , \cap , and $-$;

and let G be a boolean function involving

- Temporal functions, as just described;
- Relational operators $<$, $=$, and $>$; and
- Logical operators \wedge , \vee , and \neg .

Then, $\delta_{G, \{(I_1, V_1), \dots, (I_m, V_m)\}}(R)$, the historical derivation of R , is defined as

$$\begin{aligned}
& \delta_{G, \{(I_1, V_1), \dots, (I_m, V_m)\}}(R) \triangleq \\
& \{ u^m \mid \exists r, (r \in R \\
& \wedge \forall a, 1 \leq a \leq m, \\
& (\text{Value}(u(I_a)) = \text{Value}(r(I_a))) \\
& \wedge (\forall t, t \in \text{Valid}(u(I_a))), \\
& \exists IN_1 \dots \exists IN_m, (IN_1 \in \text{Interval}(\text{Valid}(r(I_1)))) \wedge \dots \\
& \wedge IN_m \in \text{Interval}(\text{Valid}(r(I_m))) \\
& \wedge G((I_1, IN_1), \dots, (I_m, IN_m)) \\
& \wedge t \in V_a((I_1, IN_1), \dots, (I_m, IN_m)) \\
&) \\
&) \\
& \wedge (\forall IN_1 \dots \forall IN_m, (IN_1 \in \text{Interval}(\text{Valid}(r(I_1)))) \wedge \dots \\
& \wedge IN_m \in \text{Interval}(\text{Valid}(r(I_m)))) \\
& \wedge G((I_1, IN_1), \dots, (I_m, IN_m)), \\
& V_a((I_1, IN_1), \dots, (I_m, IN_m)) \subseteq \text{Valid}(u(I_a)) \\
&)) \\
& \wedge \exists a, 1 \leq a \leq m \wedge \text{Valid}(u(I_a)) \neq \emptyset \\
& \}
\end{aligned}$$

The functions G and V_a , $1 \leq a \leq m$, are always evaluated for a specific assignment of non-disjoint intervals to attribute names I_1, \dots, I_m . G evaluates to either true or false and V_a evaluates to an element of $\wp(T)$. For a tuple $r, r \in R$, and intervals IN_b , $1 \leq b \leq m$ and $IN_b \in \text{Interval}(\text{Valid}(r(I_b)))$, we evaluate $G((I_1, IN_1), \dots, (I_m, IN_m))$ by substituting IN_b for all occurrences of I_b in G . Likewise, we evaluate $V_a((I_1, IN_1), \dots, (I_m, IN_m))$ by substituting IN_b for all occurrences of I_b in V_a . If any one of r 's attribute values has a disjoint time-stamp, there will be multiple distinct evaluations of G (and V_a) for r , one for each possible assignment of intervals to attribute names, each resulting in a value of true or false for G (and a set of chronons for V_a).

EXAMPLES.

$$\delta_{(sname \cap state) = sname, ((sname, sname), (state, sname))}(S_6) = \{ \langle ("Phil", \{1\}), ("Kansas", \{1\}) \rangle, \langle ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle \}$$

In this example, G is $(sname \cap state) = sname$ and V_1 and V_2 are both $sname$. A student tuple $s, s \in S_6$ on page 30, satisfies predicate G if the student had at least one interval of enrollment (i.e., $IN_{sname} \in Interval(Valid(s(sname)))$) during which his home state (i.e., attribute $state$) did not change (i.e., $(IN_{sname} \cap IN_{state}) = IN_{sname}$, where $IN_{state} \in Interval(Valid(s(state)))$). The new time-stamp for each attribute of a tuple that satisfies G for some assignment of intervals IN_{sname} and IN_{state} is simply the union of the IN_{sname} intervals from each assignment of intervals that satisfy G . In the first tuple in S_6 , there are three intervals, two assigned to the attribute $sname$ ($\{1\}, \{3, 4\}$) and one assigned to the attribute $state$ ($\{1, 2, 3\}$). From this tuple, we find that Phil was a resident of Kansas during his first interval of enrollment ($G((sname, \{1\}), (state, \{1, 2, 3\})) = \{1\} \cap \{1, 2, 3\} \not\leq \{1\}$) but was a resident of Kansas during only part of his second interval of enrollment ($G((sname, \{3, 4\}), (state, \{1, 2, 3\})) = \{3, 4\} \cap \{1, 2, 3\} \neq \{3, 4\}$). Hence, this tuple's attributes are assigned a time-stamp of $\{1\}$ in the resulting state. From the second tuple in S_6 we find that Phil was not a resident of Utah during his first interval of enrollment ($G((sname, \{1\}), (state, \{4, 5, 6\})) = \{1\} \cap \{4, 5, 6\} \neq \{1\}$) and lived in Utah during only part of his second interval of enrollment ($G((sname, \{3, 4\}), (state, \{4, 5, 6\})) = \{3, 4\} \cap \{4, 5, 6\} \neq \{3, 4\}$). Hence, the time-stamp for this tuple's attributes would be assigned the empty set in the resulting state except the definition of the historical derivation operator disallows tuples whose attributes all have an empty time-stamp. This tuple is therefore eliminated and does not appear in the resulting state. From the third tuple in S_6 we find that Norman was a resident of Utah during both of his intervals of enrollment ($G((sname, \{1, 2\}), (state, \{1, 2\})) = \{1, 2\} \cap \{1, 2\} \not\leq \{1, 2\}$ and $G((sname, \{5, 6\}), (state, \{5, 6\})) = \{5, 6\} \cap \{5, 6\} \not\leq \{5, 6\}$). Hence, this tuple's attributes are assigned a time-stamp of $\{1, 2, 5, 6\}$ in the resulting state. From the fourth tuple in S_6 we find that Norman was not a resident of Texas at any time during his enrollment ($G((sname, \{1, 2\}), (state, \{7, 8\})) = \{1, 2\} \cap \{7, 8\} \neq \{1, 2\}$ and $G((sname, \{5, 6\}), (state, \{7, 8\})) = \{5, 6\} \cap \{7, 8\} \neq \{5, 6\}$); this tuple is therefore eliminated from the resulting state.

$$\delta_{(sname \cap state) \neq sname \wedge (sname \cap state) \neq \emptyset, ((sname, sname \cap state), (state, sname \cap state))}(S_6) = \{ \langle ("Phil", \{3\}), ("Kansas", \{3\}) \rangle, \langle ("Phil", \{4\}), ("Utah", \{4\}) \rangle \}$$

A student tuple $s, s \in S_6$, satisfies predicate G if the student had at least one interval

of enrollment during which his home state changed. The new time-stamp for each tuple that satisfies G for some assignment of intervals IN_{sname} and IN_{state} is the union of $IN_{sname} \cap IN_{state}$ from each assignment of intervals that satisfy G . From the first tuple in S_6 we find that Phil had one interval of enrollment during which his home state changed (i.e., $\{3, 4\} \cap \{1, 2, 3\} \neq \{3, 4\}$ and $\{3, 4\} \cap \{1, 2, 3\} \neq \emptyset$). Hence, this tuple's attributes are assigned a time-stamp of $\{3, 4\} \cap \{1, 2, 3\} = \{3\}$ in the resulting state. From the second tuple in S_6 we find that Phil had one interval of enrollment during which his home state changed. Hence, this tuple's attributes are assigned a time-stamp of $\{4\}$ in the resulting state. Note that Norman does not satisfy the restriction; his home state was the same during his two periods of enrollment. Hence, the third and fourth tuples are eliminated from the resulting state. \square

Note that the historical derivation operator actually performs two functions. First, it performs a selection function on the valid-time components of a tuple's attributes. For a tuple r , if G is false when an interval from the valid-time component of each of r 's attributes is substituted for each occurrence of its corresponding attribute name in G , then the temporal information represented by that combination of intervals is not used in the calculation of the new time-stamps for r 's attributes. Secondly, the derivation operator calculates a new time-stamp for attribute I_a , $1 \leq a \leq m$, from those combinations of intervals for which G is true, using V_a . If V_1, \dots, V_m are all the same function, the tuple is effectively converted from attribute time-stamping to tuple time-stamping.

The derivation operator is necessarily complex because we allow set-valued time-stamps; it would have been less complex if we had disallowed set-valued time-stamps. Then the derivation operator could have been replaced by two simpler operators, analogous to the selection and projection operators, that would have performed tuple selection and attribute projection in terms of the valid-time components, rather than the value components, of attributes. But, as we will see in Chapter 8, disallowing set-valued time-stamps would have required that the algebra support value-equivalent tuples, which would have prevented the algebra from having several other, more highly desirable properties.

3.4 Aggregates

Aggregates allow users to summarize information contained in a relation's state. Aggregates are categorized as either *scalar aggregates* or *aggregate functions* [Snodgrass et al. 1987]. Scalar aggregates return a single scalar value that is the result of applying the aggregate to a specified attribute of a snapshot state. Aggregate functions, however, return a set of scalar values, each value the result of applying the aggregate to a specified attribute of those tuples in a snapshot state having the same values for certain attributes. Database management systems based on the relational model typically provide several aggregate operators. For example, Ingres [Stonebraker et al. 1976] provides a count, sum, average, minimum, maximum, and any aggregate operator. Ingres also provides two versions of the count, sum, and average operators, one that aggregates over all values of an attribute and one that aggregates over only the unique values of an attribute.

Several researchers have investigated aggregates in time-oriented relational databases [Ben-Zvi 1982, Jones et al. 1979, Navathe & Ahmed 1986, Snodgrass et al. 1987, Tansel et al. 1985]. Their work reflects the consensus that aggregates when applied to historical states should return not a scalar value, but a distribution of scalar values over time. Jones, et al. also introduced the concepts of *instantaneous aggregates* and *cumulative aggregates*. Instantaneous aggregates return, for each time t , a value computed only from the tuples valid at time t . Cumulative aggregates return, for each time t , a value computed from all tuples valid at any time up to and including t , regardless of whether the tuples are still valid at time t . Note that a time t has meaning only when defined in terms of the time granularity. Hence, instantaneous aggregates can be viewed as aggregates over an interval whose duration is determined by the granularity of the measure of time being used. Others have generalized the definition of instantaneous and cumulative aggregates by introducing the concept of *moving aggregation windows* [Navathe & Ahmed 1986]. For an aggregation window function w from the domain T onto the non-negative integers, an aggregate returns, for each time t , a value computed from tuples valid either at time t or at some time in the interval of length $w(t)$ immediately preceding time t . Hence, an instantaneous aggregate is an aggregate with an aggregation window function $w(t) = 0$ and a cumulative aggregate is an aggregate with an aggregation window function $w(t) = \infty$.

Klug introduced an approach to handle aggregates in the snapshot algebra [Klug 1982]. His approach makes it possible to define aggregates in a rigorous way. We use his approach to define two historical aggregate functions for our algebra:

- \widehat{A} , that calculates non-unique aggregates, and
- \widehat{AU} , that calculates unique aggregates.

These two historical aggregate functions serve as the historical counterpart of both scalar aggregates and aggregate functions.

The historical aggregate functions must contend with a variety of demands that surface as parameters (subscripts) to the functions. First, a specific aggregate (e.g., count) must be specified. Secondly, the attribute over which the aggregate is to be applied must be stated and the aggregation window function must be indicated. Finally, to accommodate partitioning, where the aggregate is applied to partitions of a historical state, a set of partitioning attributes must be given. These demands complicate the definitions of \widehat{A} and \widehat{AU} , but at the same time ensure some degree of generality to these operators.

For both definitions, let R be a historical state of m -tuples over the relation signature z with attributes $A_R = \{I_1, \dots, I_m\}$. Also let Q be a historical state with attributes A_Q , where $A_Q \subseteq A_R$. Finally, assume that we are given identifiers I_a and I_{agg} and a set of identifiers B , with the restrictions that $I_a \notin B$, $B \cup \{I_a\} \subseteq A_Q$, and $I_{agg} \notin A_Q$. If B is empty, our historical aggregate functions simply calculate a single distribution of scalar values over time for an arbitrary aggregate applied to attribute I_a of R . If B is not empty, our historical aggregate functions calculate, for each subtuple in Q formed from the attributes B , a distribution of scalar values over time for an arbitrary aggregate applied

to attribute I_a of the subset of tuples in R whose values for attributes B match the values for attributes B of the tuple in Q . Hence, B corresponds to the by-list of an aggregate function in conventional database query languages. I_{agg} is simply the name of the aggregate attribute in the resulting state. Assume, as does Klug, that for each aggregate operation (e.g., count) we have a family of scalar aggregates (e.g., Count) that performs the indicated aggregation on R (e.g., $Count_{I_1}, Count_{I_2}, \dots, Count_{I_m}$, where $Count_{I_a}, 1 \leq a \leq m$, counts the (possibly duplicate) values of attribute I_a of R). We will define our historical aggregate functions in terms of these scalar aggregates.

3.4.1 Partitioning Function

Before defining the historical aggregate functions \widehat{A} and \widehat{AU} , we define a partitioning function that will be used in their definitions. This function simply extracts from historical state R those tuples that participate in the calculation of an aggregate value for attributes B of a tuple q , $q \in Q$, at time t . The function also restricts the attribute time-stamps of selected tuples to intervals that overlap a specified aggregation window at time t .

$$\begin{aligned}
\text{Partition}(R, q, t, w, I_a, B) \triangleq & \\
& \{u^m \mid (\exists r), (r \in R \wedge \forall I, I \in B, \text{Value}(r(I)) = \text{Value}(q(I))) \\
& \quad \wedge \forall I, I \in \mathcal{A}_R, \\
& \quad (\text{Value}(u(I)) = \text{Value}(r(I))) \\
& \quad \wedge (\forall t', t' \in \text{Valid}(u(I)), \\
& \quad \exists IN, (IN \in \text{Interval}(\text{Valid}(r(I)))) \\
& \quad \wedge t - w(t) < 1 \rightarrow (IN \cap \text{Extend}(1, t) \neq \emptyset) \\
& \quad \wedge t - w(t) \geq 1 \rightarrow (IN \cap \text{Extend}(t - w(t), t) \neq \emptyset) \\
& \quad \wedge t' \in IN \\
& \quad) \\
& \quad) \\
& \quad \wedge (\forall IN, (IN \in \text{Interval}(\text{Valid}(r(I)))) \\
& \quad \wedge t - w(t) < 1 \rightarrow (IN \cap \text{Extend}(1, t) \neq \emptyset) \\
& \quad \wedge t - w(t) \geq 1 \rightarrow (IN \cap \text{Extend}(t - w(t), t) \neq \emptyset)), \\
& \quad IN \subseteq \text{Valid}(u(I)) \\
& \quad)) \\
& \quad \wedge \text{Valid}(u(I_a)) \neq \emptyset \\
& \quad \wedge \forall I, I \in B, \text{Valid}(u(I)) \neq \emptyset \\
& \quad)\}
\end{aligned}$$

where $q \in Q$, $t \in T$, and w is an aggregation window function. This function retrieves from R those tuples that have the same value components for attributes B as q and have time t , or some time in the interval of length $w(t)$ immediately preceding t , in the time-stamp of attributes B and I_a . Note that for each tuple in the resulting state, the time-stamp of attribute I_b , $1 \leq b \leq m$, is constructed from those intervals in the time-stamp of attribute I_b in the value-equivalent tuple in R that contain time t , or some time in the interval of length $w(t)$ immediately preceding t . The predicates $t - w(t) < 1 \rightarrow \dots$ and $t - w(t) \geq 1 \rightarrow \dots$ are used here to ensure that *Partition* is well-defined as *Extend* is defined only for elements in the domain T .

EXAMPLES.

$$\text{Partition}(S_6, (\), 5, 0, \text{sname}, 0) = \{ \langle ("Norman", \{5,6\}), ("Utah", \{5,6\}) \rangle \\ \langle ("Norman", \{5,6\}), ("Texas", \emptyset) \rangle \}$$

Because time 5 is specified and the aggregation window function, denoted by zero, is the constant function $w(t) = 0$, tuples are selected whose time-stamp for attribute *sname* overlaps time 5. Only the third and fourth tuples in S_6 satisfy this requirement. The partitioning function here effectively returns the tuples for those students who were enrolled in school at time 5. Note that the time-stamp of each attribute in the selected tuples has been restricted to the interval from the attribute's original time-stamp overlapping time 5, if any.

$$\text{Partition}(S_6, \langle ("Phil", \{1,3,4\}), ("Utah", \{4,5,6\}) \rangle, 5, 0, \text{sname}, \{\text{state}\}) = \\ \{ \langle ("Norman", \{5,6\}), ("Utah", \{5,6\}) \rangle \}$$

where Q is here assumed to be S_6 . Tuples are selected for those students who were enrolled in school and a resident of Phil's state (Utah) at time 5. Only the third tuple in S_6 satisfies this requirement. Although Phil was a resident of Utah at time 5, he was not enrolled in school at time 5. Hence, the second tuple in S_6 is not included in this partition.

$$\text{Partition}(S_6, \langle ("Phil", \{1,3,4\}), ("Utah", \{4,5,6\}) \rangle, 5, 1, \text{sname}, \{\text{state}\}) = \\ \{ \langle ("Phil", \{3,4\}), ("Utah", \{4,5,6\}) \rangle \\ \langle ("Norman", \{5,6\}), ("Utah", \{5,6\}) \rangle \}$$

Here tuples are selected for those students who were enrolled in school and a resident of Utah within a year ($w(t) = 1$) of time 5. Both the second and third tuples in S_6 satisfy this requirement. The second tuple in S_6 is now included in the partition because Phil was a resident of Utah and enrolled in school at time 4. \square

3.4.2 Non-unique Aggregates

The historical aggregate function \hat{A} calculates, for each tuple in Q , a distribution of scalar values over time for an arbitrary aggregate applied to attribute I_a of the subset of tuples in R whose value components for attributes B match the value components for attributes B of the tuple in Q . If B is empty, \hat{A} simply calculates a single distribution of scalar values over time for the aggregate applied to attribute I_a of R . If we let f represent an arbitrary family of scalar aggregates and w represent an aggregation window function, then the historical aggregate function \hat{A} has the following form.

$$\hat{A}_{f, w, I_a, I_{agg}, B}(Q, R)$$

EXAMPLE.

$$\hat{A}_{Count, 0, state, semester-count, \emptyset}(\hat{\pi}_{state}(S_6), S_6)$$

In this example, \hat{A} applies the aggregate operation count to attribute `state` of S_6 to compute a value for the new aggregate attribute `semester-count`. Because the aggregation window function is the constant function $w(t) = 0$, an instantaneous aggregate is computed. Also, because there are no by-list attributes (i.e., B is empty), a single distribution of scalar values over time is computed. We discuss this example further, once we have defined the historical aggregate function \hat{A} formally. \square

We now define \hat{A} on the historical states Q and R , denoted by $\hat{A}_{f, w, I_a, I_{agg}, B}(Q, R)$, as

$$\begin{aligned} \hat{A}_{f, w, I_a, I_{agg}, B}(Q, R) &\triangleq \bigcup_{t \in T} (\hat{\pi}_{B \cup \{I_{agg}\}} \\ &\{q \cup \{I_{agg} \rightarrow (x, \{t\})\} \mid q \in Q \\ &\wedge t - w(t) < 1 \rightarrow (\text{Valid}(q(I_a)) \cap \text{Extend}(1, t) \neq \emptyset \\ &\wedge \forall I, I \in B, \\ &\quad \text{Valid}(q(I)) \cap \text{Extend}(1, t) \neq \emptyset) \\ &\wedge t - w(t) \geq 1 \rightarrow (\text{Valid}(q(I_a)) \cap \text{Extend}(t - w(t), t) \neq \emptyset \\ &\wedge \forall I, I \in B, \\ &\quad \text{Valid}(q(I)) \cap \text{Extend}(t - w(t), t) \neq \emptyset) \\ &\wedge x = f_{I_a}(q, t, \text{Partition}(R, q, t, w, I_a, B)) \\ &\})) \end{aligned}$$

where $I_{agg} \rightarrow (x, \{t\})$ denotes the assignment of the aggregate value $(x, \{t\})$ to the attribute I_{agg} . If B is not empty, function \hat{A} first associates with each time t the partition of historical state Q whose tuples have t , or a time in the interval of length $w(t)$ immediately preceding

t , in the valid-time component of attributes B . For each of these partitions, \hat{A} then constructs a set of historical tuples. Each tuple in the set contains all the attributes B of a tuple q in the partition and a new aggregate attribute, I_{agg} . This new attribute's valid-time component is the time t corresponding to the partition and its value component is the scalar value returned by the aggregate f_{I_a} , when f_{I_a} is applied to the partition of R whose tuples have value components that match q 's value components for attributes B and I_a and whose valid-time components for attributes B and I_a overlap either t or the interval of length $w(t)$ immediately preceding t . Then \hat{A} performs a historical union of the resulting sets of historical tuples to produce a distribution of aggregate values over time for each tuple in Q . If B is empty, \hat{A} constructs for each time t a historical state that is either empty or contains a single tuple. If the valid-time component of attribute I_a of no tuple r in R overlaps t or the interval of length $w(t)$ immediately preceding t , then the historical state is empty. Otherwise, the historical state contains a single tuple whose valid-time component is the time t and whose value component is the scalar value returned by the aggregate f_{I_a} , when f_{I_a} is applied to the partition of R whose tuples have a valid-time component for attribute I_a that overlaps either t or the interval of length $w(t)$ immediately preceding t . Then \hat{A} performs a historical union of the resulting sets of historical tuples to produce a single distribution of aggregate values over time.

Note that a tuple and a time are passed as parameters to the scalar aggregate f_{I_a} , along with a partition of R , in the definition of \hat{A} . Although most aggregate operators can be defined in terms of a single parameter, the partition of R , the additional parameters are present because aggregates that evaluate to events or intervals, one of which is defined in Section 5.3, require them.

EXAMPLES. $\hat{A}_{Count, 0, state, semester-count, \emptyset}(\pi_{state}(S_6), S_6) = \{ \langle (1, \{3, 4, 7, 8\}) \rangle, \langle (2, \{1, 2, 5, 6\}) \rangle \}$

The function \hat{A} computes the number of states in which enrolled students resided. Because $w(t) = 0$ and the time granularity of S_6 is a semester, the resulting state represents aggregation by semester. Hence, the aggregate is in effect an instantaneous aggregate. For the interval $\{1, 2\}$, there were two states (Kansas in the first tuple and Utah in the third tuple). For the interval $\{3, 4\}$, there was one state (Kansas in the first tuple at time 3 and Utah in the second tuple at time 4). For the interval $\{5, 6\}$, there also was only one state (Utah), but it appeared in both the second and the third tuples. It was counted twice because the scalar aggregates embedded within \hat{A} aggregate over duplicate values. For the interval $\{7, 8\}$, there was only one state (Texas in the fourth tuple).

$\hat{A}_{Count, 1, state, year-count, \emptyset}(\pi_{state}(S_6), S_6) = \{ \langle (1, \{8, 9\}) \rangle, \langle (2, \{1, 2, 3, 4, 5, 6\}) \rangle, \langle (3, \{7\}) \rangle \}$

Again, \hat{A} computes the number of states in which enrolled students resided, but now $w(t) = 1$. Hence, the resulting state now represents aggregation by year (assuming two semesters per year). Although nine does not appear in the time-stamp of attribute **state** in any tuple in S_6 , a count of one is recorded at time 9 because a tuple, the fourth tuple in S_6 , falls into the aggregation window at time 9.

$$\hat{A}_{Count, \infty, state, total-count, \emptyset}(\hat{\pi}_{state}(S_6), S_6) = \{ \langle (2, \{1, 2, 3\}) \rangle, \\ \langle (3, \{4, 5, 6\}) \rangle, \\ \langle (4, \{7, 8, \dots\}) \rangle \}$$

Now, with $w(t) = \infty$, \hat{A} computes a cumulative aggregate of the number of states in which enrolled students resided.

$$\hat{A}_{Count, 0, sname, students, \{state\}}(S_6, S_6) = \{ \langle ("Kansas", \{1, 2, 3\}), (1, \{1, 2, 3\}) \rangle \\ \langle ("Utah", \{1, 2, 4, 5, 6\}), (1, \{1, 2, 4\}) \rangle \\ \langle ("Utah", \{1, 2, 4, 5, 6\}), (2, \{5, 6\}) \rangle \\ \langle ("Texas", \{7, 8\}), (1, \{7, 8\}) \rangle \}$$

Here, \hat{A} computes the instantaneous aggregate of the number of enrolled students who resided in each state. In effect, the aggregate is computed for each subset of tuples in S_6 having the same value for the attribute **state**. For example, the first tuple is computed by selecting all the tuples in S_6 with a state of Kansas and then performing the aggregate on this (smaller) set. \square

3.4.3 Unique Aggregates

The function \hat{A} allows its embedded scalar aggregates to aggregate over duplicate attribute values. We now define a historical aggregate function \widehat{AU} , identical to \hat{A} with one exception; it restricts its embedded scalar aggregates to aggregation over unique attribute values. We define \widehat{AU} on the historical states Q and R , denoted by $\widehat{AU}_{f, w, I_a, I_{agg}, B}(Q, R)$, as

$$\begin{aligned}
\widehat{AU}_{f, w, I_a, I_{agg}, B}(Q, R) \triangleq \bigcup_{t \in T} (\pi_{B \cup \{I_{agg}\}} (& \\
\{q \cup \{I_{agg} \rightarrow (x, \{t\})\} \mid q \in Q & \\
\wedge t - w(t) < 1 \rightarrow (\text{Valid}(q(I_a)) \cap \text{Extend}(1, t) \neq \emptyset & \\
\wedge \forall I, I \in B, & \\
\text{Valid}(q(I)) \cap \text{Extend}(1, t) \neq \emptyset) & \\
\wedge t - w(t) \geq 1 \rightarrow (\text{Valid}(q(I_a)) \cap \text{Extend}(t - w(t), t) \neq \emptyset & \\
\wedge \forall I, I \in B, & \\
\text{Valid}(q(I)) \cap \text{Extend}(t - w(t), t) \neq \emptyset) & \\
\wedge x = f_{I_a}(q, t, \delta_{true, t}(\pi_{I_a}(\text{Partition}(R, q, t, w, I_a, B)))) & \\
})) &
\end{aligned}$$

This definition differs from that of \widehat{A} only in that the historical projection on attribute I_a of $\text{Partition}(\dots)$ followed by the historical derivation eliminates duplicate values of the aggregated attribute before the scalar aggregation is performed.

EXAMPLE. $\widehat{AU}_{Count, 0, state, semester-count, 0}(\pi_{state}(S_6), S_6) = \{ \langle (1, \{3, 4, 5, 6, 7, 8\}) \rangle, \langle (2, \{1, 2\}) \rangle \}$

This state differs from the non-unique variant only during the interval $\{5, 6\}$. Here, Utah is correctly counted only once, even though there are two tuples valid during this interval with a state of Utah. \square

3.4.4 Expressions in Aggregates

The functions \widehat{A} and \widehat{AU} allow expressions to be aggregated and support aggregation by arbitrary expressions. Let $Eaggregate$ be an arbitrary expression involving u historical aggregate functions. Also, assume that the v^{th} historical aggregate function applies the scalar aggregate f_v to attribute I_{av} , where the aggregation window function is w_v , and the partitioning attributes are B_v . Then the definition of \widehat{A} , now denoted by

$$\widehat{A}_{f_1, w_1, I_{a_1}, B_1, \dots, f_u, w_u, I_{a_u}, B_u, I_{agg}, Eaggregate}(Q, R)$$

is constructed from the definition of \widehat{A} above simply by substituting $x = Eaggregate'$ for $x = f_{N_a}(\dots)$. $Eaggregate'$ is $Eaggregate$ where each reference to the v^{th} aggregate has been replaced by the expression $f_{v, I_{av}}(q, t, \text{Partition}(R, q, t, w_v, I_{av}, B_v))$. With these changes, \widehat{A} allows expressions to be aggregated. \widehat{AU} can be modified similarly.

If \hat{A} and \widehat{AU} are to support aggregation by arbitrary expressions, changes must be made to the definitions of *Partition*, \hat{A} , and \widehat{AU} given above. First, let $Evalue_l$, $1 \leq l \leq u$, be an expression involving attribute names in A_Q . $Evalue_l$ is evaluated for a tuple r in R (or a tuple q in Q) by substituting the value components of the attributes of r (or q) for all occurrences of their corresponding attribute names in $Evalue_l$. Secondly, let B be the set of attributes names that appear in at least one $Evalue_l$, $1 \leq l \leq u$. Then the definition of *Partition*, now denoted by

$$\text{Partition}(R, q, t, w, I_a, B, \{Evalue_1, \dots, Evalue_u\})$$

is constructed from the definition of *Partition* above simply by substituting the predicate $\forall l, 1 \leq l \leq u, Evalue_l(r) = Evalue_l(q)$ for the predicate $\forall I, I \in B, Value(r(I)) = Value(q(I))$. The definition of \hat{A} , now denoted by

$$\widehat{AU}_f, w, I_a, I_{agg}, B, \{Evalue_1, \dots, Evalue_u\}(Q, R)$$

is constructed from the definition of \hat{A} above simply by adding $\{Evalue_1, \dots, Evalue_u\}$ as an additional parameter of the partitioning function. \widehat{AU} can be modified similarly. With these changes, \hat{A} and \widehat{AU} support aggregation by arbitrary expressions.

3.5 Preservation of the Value-equivalence Property

Theorem 3.1 *The operators \cup , \cap , \hat{x} , δ , π , δ , \hat{A} , and \widehat{AU} all preserve the value-equivalence property of historical states.*

PROOF. For the operators \cup , \cap , \hat{x} , δ , and δ we show that the contrapositive of the theorem holds, that is, if there are value-equivalent tuples in an operator's output relation, then there are value-equivalent tuples in at least one of its input relations. For the operators π , \hat{A} , and \widehat{AU} , we show by contradiction that there cannot be value-equivalent tuples in their output relations.

Case 1. \cup . Assume that $Q \cup R$ contains at least two value-equivalent tuples. From the definition of \cup , each tuple in $Q \cup R$ has a value-equivalent tuple in Q , R , or both. If two value-equivalent tuples \hat{u}_1 and \hat{u}_2 in $Q \cup R$ do not have a value-equivalent tuple in R , then both are tuples in Q . Similarly, if they do not have a value-equivalent tuple in Q , then both are tuples in R . If they have a value-equivalent tuple in both Q and R , then each was constructed from a value-equivalent tuple in Q and a value-equivalent tuple in R . If both \hat{u}_1 and \hat{u}_2 had been constructed from the same tuple in Q and the same tuple in R , then \hat{u}_1 and \hat{u}_2 would be, by definition, the same tuple. Hence, they were constructed from different value-equivalent tuples in Q , R , or both.

Case 2. \cap . Assume that $Q \cap R$ contains at least two value-equivalent tuples. From the definition of \cap , each tuple in $Q \cap R$ has a value-equivalent tuple in Q but not in R or a value-equivalent tuple in both Q and R . If two value-equivalent tuples \hat{u}_1 and \hat{u}_2 in

$Q - R$ do not have a value-equivalent tuple in R , then both are tuples in Q . If they have a value-equivalent tuple in both Q and R , then each was constructed from a value-equivalent tuple in Q and a value-equivalent tuple in R . If both \hat{u}_1 and \hat{u}_2 had been constructed from the same tuple in Q and the same tuple in R , then \hat{u}_1 and \hat{u}_2 would be, by definition, the same tuple. Hence, they were constructed from different value-equivalent tuples in Q , R , or both.

Case 3. \hat{x} . Assume that $Q \hat{\times} R$ contains at least two value-equivalent tuples. From the definition of \hat{x} , each tuple in $Q \hat{\times} R$ is constructed from a tuple in Q and a tuple in R . If two value-equivalent tuples \hat{u}_1 and \hat{u}_2 in $Q \hat{\times} R$ had been constructed from the same tuple in Q and the same tuple in R , then \hat{u}_1 and \hat{u}_2 would be, by definition, the same tuple. Hence, they were constructed from different value-equivalent tuples in Q , R , or both.

Case 4. $\hat{\sigma}$. Assume that $\hat{\sigma}_F(R)$ contains at least two value-equivalent tuples. From the definition of $\hat{\sigma}$, each tuple in $\hat{\sigma}_F(R)$ is a tuple in R . Hence, any two value-equivalent tuples in $\hat{\sigma}_F(R)$ are also tuples in R .

Case 5. $\hat{\pi}$. Assume that $\hat{\pi}_X(R)$ contains at least two value-equivalent tuples. For any two such tuples there will be at least one time that appears in the time-stamp of an attribute of one tuple but not the other tuple; otherwise, they would be identical. Hence, let \hat{u}_1 and \hat{u}_2 be two value-equivalent tuples in $\hat{\pi}_X(R)$ such that there is a time t in the time-stamp of attribute I , $I \in X$, of \hat{u}_1 but not \hat{u}_2 . From the first clause of the definition of $\hat{\pi}$, there is a tuple r , $r \in R$, that has t in the time-stamp of attribute I and the same value for attributes X as \hat{u}_1 . But, from the second clause of the definition, the time-stamp of attribute I of tuple r is a subset of the time-stamp of attribute I of \hat{u}_2 , as r also has the same value for attributes X as \hat{u}_2 . Hence, t is in the time-stamp of attribute I of \hat{u}_2 , contradicting the assumption that t is in the time-stamp of attribute I of \hat{u}_1 but not \hat{u}_2 . Similarly, we arrive at a contradiction if we assume that there is a time t in the time-stamp of attribute I of \hat{u}_2 but not \hat{u}_1 . Hence, \hat{u}_1 and \hat{u}_2 have identical attribute time-stamps, which implies that they are the same tuple, contradicting the assumption that $\hat{\pi}_X(R)$ contains at least two value-equivalent tuples. Note that the output relation of $\hat{\pi}$, unlike the output relations of $\hat{\cup}$, $\hat{-}$, $\hat{\times}$, and $\hat{\sigma}$, would not contain value-equivalent tuples even if there were value-equivalent tuples in its input relation.

Case 6. δ . Assume that $\delta_{G, \{(I_1, V_1), \dots, (I_m, V_m)\}}(R)$ contains at least two value-equivalent tuples, \hat{u}_1 and \hat{u}_2 . From the definition of δ , each tuple in $\delta_{G, \{(I_1, V_1), \dots, (I_m, V_m)\}}(R)$ is constructed from one value-equivalent tuple in R . If \hat{u}_1 and \hat{u}_2 were constructed from the same value-equivalent tuple r , $r \in R$, then they would be the same tuple, as δ requires not only that every time t in the time-stamp of attribute I_a , $1 \leq a \leq m$, of either \hat{u}_1 or \hat{u}_2 be in $V_a(\dots)$ and satisfy $G(\dots)$ for some assignment of intervals from the time-stamps of r 's attributes to attribute names but that $V_a(\dots)$ be a subset of the time-stamp of attribute I_a of both \hat{u}_1 and \hat{u}_2 . Hence, \hat{u}_1 and \hat{u}_2 were constructed from different value-equivalent tuples in R .

Case 7. \hat{A} . Assume that $\hat{A}_{f, w, I_a, I_{agg}, B}(Q, R)$ contains at least two value-equivalent tuples. From Case 1 above, if $\hat{A}_{f, w, I_a, I_{agg}, B}(Q, R)$ contains value-equivalent tuples, then the input relation to \hat{A} 's outermost $\hat{\cup}$ operator contains value-equivalent tuples. But,

this relation is the output of π , whose output relation was shown in Case 5 above never to contain value-equivalent tuples. Hence, our assumption that $\hat{A}_{f, w, I_a, I_{agg}, B}(Q, R)$ contains at least two value-equivalent tuples is contradicted.

Case 8 \bar{AU} . Simply replace \hat{A} with \bar{AU} in Case 7. ■

3.6 Additional Aspects of the Algebra

We defined eight algebraic operators in Section 3.3. Yet, there are other operators that can exist harmoniously with these eight operators. For example, historical intersection ($\hat{\cap}$), Θ -join ($\hat{\Theta}$), natural join ($\hat{\bowtie}$), and quotient ($\hat{\div}$) all can be defined in terms of the six operators \cup , $\bar{\cup}$, \hat{x} , δ , π , and δ . Also, the historical rollback operator ($\hat{\rho}$), defined in Chapter 4, serves to generalize the algebra to handle temporal relation states incorporating both valid time and transaction time.

Historical intersection can be defined in an identical fashion to its snapshot counterpart. Definition of the historical version of intersection is straightforward only because we took care when defining the historical version of difference to ensure its compatible with definition of intersection in terms of difference. If we let Q and R be snapshot states of m -tuples over the relation signature z with attributes $A = \{I_1, \dots, I_m\}$, then $Q \cap R$ is defined as [Ullman 1982]

$$Q \cap R \triangleq Q - (Q - R).$$

Now, let Q and R be historical states, rather than snapshot states. Then, $Q \hat{\cap} R$, the historical intersection of Q and R , is defined as

$$Q \hat{\cap} R \triangleq Q \bar{\cup} (Q \bar{\cup} R).$$

Θ -join also can be defined in an identical fashion to its snapshot counterpart. Definition of the historical version of Θ -join is straightforward because its definition involves only selection and cartesian product, two operators whose historical versions are themselves defined in an identical fashion to their snapshot counterparts. If we let Q be a historical state of m_1 -tuples on the relation signature z_Q with attributes $A_Q = \{I_{Q,1}, \dots, I_{Q,m_1}\}$ and R be a historical state of m_2 -tuples on the relation signature z_R with attributes $A_R = \{I_{R,1}, \dots, I_{R,m_2}\}$, where $A_Q \cap A_R = \emptyset$, then the Θ -join of Q and R is defined as [Ullman 1982]

$$Q \underset{I_{Q,u} \Theta I_{R,v}}{\hat{\Theta}} R \triangleq \sigma_{I_{Q,u} \Theta I_{R,v}}(Q \times R),$$

where, $1 \leq u \leq m_1$, $1 \leq v \leq m_2$, and $I_{Q,u}$ and $I_{R,v}$ are Θ -comparable.

Now let Q and R be historical states, rather than snapshot states. Then the historical Θ -join of Q and R is defined as

$$Q \bowtie_{I_{Q,u}, I_{R,v}} R \triangleq \delta_{r_{Q,u} \in I_{Q,v}} (Q \times R).$$

Historical natural join and quotient, unlike historical difference and Θ -join, can't be defined simply by substituting historical operators for snapshot operators in the definition of their snapshot counterparts [Ullman 1982], because both involve projection, an operation whose semantics in the historical algebra is substantially different from its semantics in the snapshot algebra. Small, but important, changes must be made to the definitions to handle properly the temporal dimension. Let $A_Q = \{ I_{Q,1}, \dots, I_{Q,m_1} \}$, $A_R = \{ I_{R,1}, \dots, I_{R,m_2} \}$, and $A_{QR} = \{ I_1, \dots, I_m \}$. Also let Q be a snapshot state of (m_1+m) -tuples on the relation signature z_Q with attributes $A_Q \cup A_{QR}$ and R be a snapshot state of (m_2+m) -tuples on the relation signature z_R with attributes $A_R \cup A_{QR}$. Hence, the attributes A_{QR} are common to Q and R . Rather than rename attributes, we simply refer to the common attributes in Q and R as $Q.I_u$ and $R.I_u$, $1 \leq u \leq m$, respectively, for notational convenience. Then $Q \bowtie R$, the natural join of Q and R , is defined as [Ullman 1982]

$$Q \bowtie R \triangleq \pi_{A_Q \cup \{Q.I_1, \dots, Q.I_m\} \cup A_R} (\sigma_{Q.I_1=R.I_1 \wedge \dots \wedge Q.I_m=R.I_m} (Q \times R)).$$

Now let Q and R be historical states, rather than snapshot states. If we were to simply replace snapshot operators in the above definition with their historical counterparts, \bowtie would retain the valid time assigned to attributes A_{QR} in Q but not in R , because the projection somewhat arbitrarily keeps the common attributes from Q and not from R . Similarly, if we were also to replace references to $Q.I_u$, $1 \leq u \leq m$, in the projection operator with references to $R.I_u$, \bowtie would retain the valid time assigned to attributes A_{QR} in R but not in Q . Retention of the valid time assigned to attributes A_{QR} in both Q and R , however, seems more appropriate. Hence, we define $Q \bowtie R$, the historical natural join of Q and R , as

$$\begin{aligned} Q \bowtie R \triangleq & \pi_{A_Q \cup \{Q.I_1, \dots, Q.I_m\} \cup A_R} (\\ & \delta_{\text{true}, ((I_{Q,1}, I_{Q,1}), \dots, (I_{Q,m_1}, I_{Q,m_1}), (Q.I_1, Q.I_1 \cup R.I_1), \dots, (Q.I_m, Q.I_m \cup R.I_m), \\ & (I_{R,1}, I_{R,1}), \dots, (I_{R,m_2}, I_{R,m_2}), (R.I_1, R.I_1), \dots, (R.I_m, R.I_m))} (\\ & \delta_{Q.I_1=R.I_1 \wedge \dots \wedge Q.I_m=R.I_m} (Q \hat{\times} R))). \end{aligned}$$

The δ operator is introduced to compute the valid-time component of attributes in the resulting historical state common to both Q and R . Here, we use union semantics to retain, for each attribute common to Q and R , the valid time assigned to the attribute in both relation states. We can just as easily define other historical variations of natural join using either intersection or difference semantics. Note that the new time-stamps for attributes $R.I_1, \dots, R.I_m$ are arbitrary as these attributes are discarded by the projection operator.

To define quotient, let S be a snapshot state of $(m_1 + m_2)$ -tuples on the relation signature z_S with attributes $A_Q \cup A_R$ and R be a snapshot state of m_2 -tuples on the relation signature z_R with attributes A_R , where $A_Q = \{ I_{Q,1}, \dots, I_{Q,m_1} \}$ and $A_R = \{ I_{R,1}, \dots, I_{R,m_2} \}$.

Then, the quotient of S divided by R ($S + R$) intuitively is the maximal subset Q of $\pi_{A_Q}(S)$ such that $Q \times R$ is contained in S [Maier 1983]. $S + R$ is defined as [Ullman 1982]

$$S + R \triangleq \pi_{A_Q}(S) - \pi_{A_Q}((\pi_{A_Q}(S) \times R) - S).$$

Now let S and R be historical states, rather than snapshot states. If we were to simply replace snapshot operators in the above definition with their historical counterparts, $\dot{+}$ would not place the same restrictions on the attribute time-stamps of tuples in Q that it places on the tuples' attribute values. The operator would require that each tuple in $Q \dot{\times} R$ have a value-equivalent tuple in S , but it would not require that the attribute time-stamps of a tuple in $Q \dot{\times} R$ be contained in the attribute time-stamps of its value-equivalent tuple in S . Hence, we propose a definition of $\dot{+}$ that places the same restrictions on the attribute time-stamps of tuples in Q that it places on the tuples' attribute values. If we let the historical quotient of S divided by R ($S \dot{+} R$) be the maximal temporal contents of $\pi_{A_Q}(S)$ such that $Q \dot{\times} R$ is contained temporally in S , then $S \dot{+} R$ is defined as

$$S \dot{+} R \triangleq \pi_{A_Q}(S) - \pi_{A_Q}(U \cup \delta_{I_{R,1} \neq 0 \vee \dots \vee I_{R,m_2} \neq 0, \{(I_{Q,1}, T), \dots, (I_{Q,m_1}, T), (I_{R,1}, I_{R,1}), \dots, (I_{R,m_2}, I_{R,m_2})\}}(U))$$

where $U = (\pi_{A_Q}(S) \dot{\times} R) - S$.

The additional restriction introduced by the δ clause ensures that no tuple in $S \dot{+} R$ can combine with a tuple in R to produce a tuple whose attribute time-stamps are not contained in the attribute time-stamps of its value-equivalent tuple in S .

EXAMPLES. Assume that we are given the historical state S_6 from page 30 over the relation signature *Enrollment* with the attributes {*sname*, *state*}, duplicated below.

$$\begin{aligned} & \{ \langle ("Phil", \{1, 3, 4\}), ("Kansas", \{1, 2, 3\}) \rangle, \\ & \quad \langle ("Phil", \{1, 3, 4\}), ("Utah", \{4, 5, 6\}) \rangle, \\ & \quad \langle ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\ & \quad \langle ("Norman", \{1, 2, 5, 6\}), ("Texas", \{7, 8\}) \rangle \} \end{aligned}$$

If we are given the following historical state S_7 with attribute {*state*},

$$S_7 = \{ \langle ("Utah", \{5\}) \rangle, \langle ("Texas", \{7, 8\}) \rangle \}$$

then

$$S_6 \dot{+} S_7 = \{ \langle ("Norman", \{1, 2, 5, 6\}) \rangle \}.$$

If, however, we are given

$$S_6 = \{ \langle ("Utah", \{5\}), \\ \langle ("Texas", \{7, 8, 9\}) \rangle \} \}$$

then

$$S_6 \dot{\div} S_8 = \emptyset.$$

In the first example, although Phil lived in Utah at time 5, he was not included in $S_6 \dot{\div} S_7$ because he did not reside in Texas at times 7 and 8. In the second example, neither Phil nor Norman were included in $S_6 \dot{\div} S_8$ because neither resided in Texas at time 9. The δ clause ensured that Norman was excluded even though he lived in Utah at time 5 and in Texas at times 7 and 8. \square

In addition to defining the eight operators in Section 3.3, we restricted the valid-time component of attributes to elements from the domain $\wp(T)$. By so doing, we were able to define all operations on attribute time-stamps in terms of the standard operations from set theory. We can eliminate the restriction and allow time-stamps to have a more complex structure without difficulty. For example, we could allow the valid-time component of attributes to be an element from, or even a subset of, $\wp(T) \times \wp(T)$. We need only redefine the functions *First*, *Last*, and *Interval* to handle time-stamps of the new form and replace each set operation on time-stamps with an equivalent operation for the new time domain. In this way, our algebra could support either periodicity [Lorentzos & Johnson 1987A] or multi-dimensional time-stamps [Gadia & Yeung 1988].

We also restricted the value component of attributes to atomic elements from a value domain. Several of the other historical algebras that have been proposed allow set-valued attributes [Clifford & Croker 1987, Gadia 1986, Tansel 1986]. Their purpose in allowing set-valued attributes is to model real-world relationships more naturally and to eliminate the need to replicate data among tuples. These algebras only allow one level of nesting. Hence, while they can model the relationship between students and courses without replication of data, they can't model the relationships among students, courses, and grades without replication of data. Several proposals have already been presented for extending the snapshot algebra to support non-first-normal-form relations with an arbitrary level of nesting [Ozsoyoglu et al. 1987, Roth et al. 1984, Schek & Scholl 1986]. Hence, rather than complicate the semantics of our algebra by allowing set-valued attributes, we propose extending our algebra to support non-first-normal-form historical relations with an arbitrary level of nesting using an approach similar to the one Schek and Scholl used to extend the snapshot algebra. Then, we could define both relation states and operations on states recursively. At each recursively defined level, an attribute could take on an atomic value from a value domain or a structured value from a domain of historical relation states. Our semantics, however, would be left unchanged, simply embedded in the new structure.

We leave these last two extensions to future work.

3.7 Summary

In this chapter we have extended the snapshot algebra to support valid time by defining a historical algebra. Definition of the algebra required that we introduce only one type of object, the historical relation. A historical relation was defined in terms of its scheme (i.e., relation class and signature) and the set of states that it can assume. Valid time was accommodated by assigning set-valued time-stamps to attributes. Also, 12 operations on historical states were defined.

- Nine of the operations have counterparts in the snapshot algebra: union (\cup), difference ($-$), cartesian product (\times), selection (δ), projection (π), intersection (\cap), Θ -join (\bowtie), natural join (\bowtie), and quotient (\div).
- Historical derivation (δ) effectively performs selection and projection on the valid-time component of attributes by replacing the time-stamp of each attribute of selected tuples with a new time-stamp.
- Aggregation (\hat{A}) and unique aggregation (\widehat{AU}) serve to compute a distribution of aggregate values over time for a collection of tuples.

After defining the algebra, we discussed ways to extend the algebra to allow time-stamps with a complex structure and to support non-first-normal-form historical relations.

This chapter makes two contributions. The primary contribution is the algebra itself. By making appropriate design decisions (i.e., associating valid time with attributes rather than with tuples, representing valid time as a set of chronons, and requiring that the value component of attributes be atomic-valued), we were able to define a historical algebra that is a relatively straightforward extension of the snapshot algebra. As we show in Chapter 8, the algebra also has a collection of desirable properties satisfied in concert by no other historical algebra. The second contribution is the formal definition of the type of object and the operations on object instances allowed in the algebra. Formal definitions make the algebra unambiguous. They also are the basis for proving that the algebra has the expressive power of calculus-based query languages and they may be used to prove various implementations of these languages correct. In Chapter 5 we show that the algebra defined here has the expressive power of the temporal query language TQuel.

We found definition of a historical algebra to be a surprisingly difficult task. Although it is relatively easy to define an algebra that has a single property, it is much more difficult to define an algebra that has many desirable properties. We found that many subtle issues arise when attempting to define an algebra that satisfies several design goals. Also, all desirable properties of historical algebras are not compatible, as we show in Chapter 8. Hence, the best that can be hoped for is not an algebra with all possible desirable properties but an algebra with a maximal subset of the most desirable properties. The historical algebra defined here has what we consider to be the most desirable properties of a historical algebra. In Chapter 8, we review the historical algebras proposed by others, identify a set

of properties desirable of historical algebras, and compare our algebra and those proposed by others, using the properties as evaluation criteria.

In the next chapter, we extend both the snapshot algebra and our historical algebra to handle transaction time.

Chapter 4

Adding Transaction Time

In the previous chapter we extended the snapshot algebra to handle valid time by defining a historical algebra. We did not consider, however, any extension of the snapshot algebra or our historical algebra to support transaction time. In this chapter we describe an approach for adding transaction time to both.

Transaction time concerns the storage of information in a database. A database's *scheme* describes the structure of the database; the *contents* of the database must adhere to that structure [Date 1976, Ullman 1982]. *Scheme evolution* refers to changes to the database's scheme over time; *contents evolution* refers to changes to the database's contents over time. Hence, a model of transaction time needs to support both scheme evolution and contents evolution. Conventional DBMS's retain only the current contents of a database and allow only one scheme to be in force at a time, requiring *restructuring* (also termed *logical reorganization* [Socquet & Goldberg 1979]) when the scheme is changed. This model of transaction time, although adequate for databases containing only snapshot and historical relations, is inadequate for databases containing rollback and temporal relations. As we saw in Chapter 1, rollback and temporal relations must retain past information to support rollback operations. To model transaction time in databases containing relations of all four classes, we define an algebraic language for database query and update that allows past database contents to be retained and accommodates multiple schemes, each in effect for an interval in the past.

Several benefits accrue from defining a language that extends the algebras to support transaction time. Although not available in the algebras, the action of update is available in the language, allowing the language to be the executable form to which update operations in a calculus-based language (e.g., append, delete, replace in Quel [Held et al. 1975] or TQuel [Snodgrass 1987]) can be mapped. If these operations in the calculus are formalized, the mapping can be proven correct. Secondly, update optimizations, analogous to the retrieval optimizations that have been studied extensively [Smith & Chang 1975], can now be investigated in a rigorous fashion. A third benefit is that the *database state* (i.e., the database's scheme and contents), and its evolution, are now placed on a formal basis. In particular, the domain of database states and the change to each state effected by each

update operation are defined. Of course, actual implementations will vary considerably in the physical structures used to encode a database state on secondary storage. However, the existence of a formal definition of database state allows rigorous statements to be made concerning the correctness of those structures and the information content of the database.

Another benefit accrues from our approach for adding transaction time to the algebras. Our approach is general; it depends on no specific technique for adding valid time to the snapshot algebra. Rather, it is compatible with any such technique. Hence, our approach can be applied to any historical algebra to yield an algebraic language for the query and update of temporal databases.

4.1 Approach

The key aspect of the relational algebra is its definition of snapshot state, which models reality at one time. Similarly, the key aspect of our historical algebra is its definition of historical state, which models reality over an interval. Neither algebra, however, is adequate to model changes in database state because neither has update semantics. We now want to extend the algebras to support both aspects of transaction time: evolution of a database's scheme and evolution of its contents.

We saw in Chapter 1 that a relation's structure cannot be defined in terms of the relation's attributes alone; it must also be defined in terms of the relation's class. Hence, we define a relation's scheme to be a pair consisting of the relation's class and a function, which we refer to as the relation's *signature*, that maps the relation's attribute names onto their value domains. (If the identification of primary keys is desirable, this would also properly go into the signature.) The relation's contents, which we refer to as the relation's *state*, always must be consistent with both the relation's class and the relation's signature.

Our model of transaction time is predicated on two assumptions. First, we assume that a database may contain snapshot, rollback, historical, and temporal relations. Second, we assume that the class and signature, as well as the contents, of each relation in the database may change over time. For example, a relation defined initially as a snapshot relation could be changed to be a historical, rollback, or temporal relation. Later, it could be changed to be a snapshot relation once again.

A model of transaction time in a database containing relations of all four classes, must maintain, for each relation, its current class, signature, and state. The model also must retain, for each relation, its signature and state for those intervals during which its class was either rollback or temporal. Hence, we define a relation to be a triple consisting of a sequence of classes, a sequence of signatures, and a sequence of states, all ordered by transaction number. The class sequence records the relation's current class and intervals when the relation's class was either rollback or temporal. Similarly, the signature and state sequences record the relation's current signature and state and all changes in signature and state during intervals when the relation's class was either rollback or temporal. We also define a database state to be a function from identifiers (i.e., relation names) to relations.

Finally, we define a database to be an ordered pair whose first component is a database state and whose second component is the transaction number of the most recently committed transaction on the database.

When transaction time is supported by a DBMS, a means of accessing states other than a relation's current state must be included. A relation's past states when its class was either rollback or temporal always must be accessible via rollback operations. We define a new algebraic operator called *rollback* to make past states available in the algebras. Fortunately, rollback, like the other algebraic operators, has no side-effects, so it is easily incorporated into the algebras.

Extension of the algebras to include update semantics, however, poses a fundamental problem. The algebras by definition are side-effect-free, but the essential aspect of a database transaction is solely its side-effect of changing the database. One awkward but perhaps feasible solution is to add the database as a parameter to every operator in the algebras. We adopt a different strategy, leaving the basic structure of the algebras intact, and instead inserting them into a structure of *commands* that provide the needed side-effects. Hence, what we are proposing is a *language* with the algebras, augmented with rollback operators, as significant components. In doing so, we preserve all the properties of the two algebras (e.g., commutativity of select, distributivity of select over join), permitting the full application of algebraic optimizations in expression evaluation.

We define four commands for database update: *define_relation*, *modify_relation*, *destroy*, and *rename_relation*. The *define_relation* command assigns a new class and signature, along with the empty snapshot or historical state, to an undefined relation. The *modify_relation* command changes the current class, signature, and state of a defined relation. The *destroy* command is the counterpart of the *define_relation* command. It either physically or logically deletes from the database the current class, signature, and state of a relation, depending on the relation's class when the command is executed. The *rename_relation* command binds the current class, signature, and state of a relation to a new identifier. We assume that these commands execute in the context of a single, previously created database. Hence, no commands are necessary to create or delete the database. Since we are considering modeling transaction time from a functional, rather than from a performance, viewpoint, commands affecting access methods, storage mechanisms, or index maintenance are also not relevant.

Allowing a database's scheme, as well as its contents, to change increases the complexity of our language. If we allow the database's scheme to change, an algebraic expression that is semantically correct for the database's scheme when one command executes may not be semantically correct for the database's scheme when another command executes. We now need a mechanism for identifying semantically incorrect algebraic expressions relative to the database's scheme when each command executes and a way of ensuring that the scheme and contents of the database state resulting from the command's execution are compatible. To identify semantically incorrect expressions, we introduce a *semantic type system* and augment all commands to do type-checking.

Finally, we encapsulate commands within a system of transactions to provide for both

single-command and multiple-command transactions. A multiple-command transaction, like a single-command transaction, is treated as an atomic update operation, whether it changes one relation or several relations. Transactions are specified by the keywords `begin_transaction` and either `commit_transaction` or `abort_transaction`, the later depending on whether the transaction commits or aborts.

Summarizing these changes, we add

- the scheme (i.e., class and signature) to the formal definition of database state;
- the capability to retain selected information about a relation's past by defining a relation as a sequence of classes, a sequence of signatures, and a sequence of states;
- a rollback operator to the algebras to access past states;
- four commands to change the database state;
- a semantic type system to identify semantically incorrect algebraic expressions and enforce consistency constraints between the scheme and contents of the database; and
- a system of transactions to provide for single-command and multiple-command transactions.

The result is an algebraic language that supports both aspects of transaction time: evolution of a database's scheme and evolution of its contents.

This language was designed to satisfy several other objectives as well. First, the language subsumes the expressive power of the snapshot algebra. For every expression in the snapshot algebra, there is an equivalent expression in the language. Second, the language subsumes the expressive of our historical algebra. For every expression in our historical algebra, there is an equivalent expression in the language. Third, the language ensures that all data stored in a relation when its class was either rollback or temporal are retained permanently and are accessible via a rollback operator, even after the relation is logically deleted from the database. Fourth, commands change only a relation's class, signature, and state current at the start of a transaction. Past data that are retained to support rollback operations, once saved, are never changed. Hence, the language accommodates implementations that use write-once-read-many (WORM) optical disk to store non-current class, signature, and state information.

We employ denotational semantics to define the semantics of the language, due to its success in formalizing operations involving side-effects, such as assignment, in programming languages [Gordon 1979, Stoy 1977]. In defining the semantics of commands and algebraic operators, we have favored simplicity of semantics at the expense of efficient direct implementation. The language would be inefficient, in terms of storage space and execution time, if mapped directly into an implementation. However, the semantics do not preclude more efficient implementations using optimization strategies for both storage and retrieval of information. In Section 4.4, we review briefly some of the techniques for

efficient implementation, compatible with our semantics, that have been proposed by others. We also, without loss of generality, assume that transactions are executed sequentially in a single-user environment. Our approach applies equally to environments that permit the concurrent execution of transactions as long as their concurrency control mechanisms induce a serialization of transactions.

Our language for supporting the above extensions will be the topic of the next section. Additional aspects of the rollback operators are discussed briefly in Section 4.3. Section 4.4 will review related work and compare our approach with those of others.

4.2 The Language

In this section we provide the syntax and denotational semantics of our language for database query and update. In denotational semantics, a language is described by assigning to each language construct a *denotation* – an abstract entity that models its meaning [Gordon 1979, Scott 1976, Stoy 1977, Strachey 1966]. We chose denotational semantics to define our language because denotational semantics combines a powerful descriptive notation with rigorous mathematical theory [Gordon 1979], permitting the precise definition of database state. First, we define the syntax of the language. Then we define the language's semantic domains and a semantic type system for expressions. Finally, we define the semantic functions that map the language constructs onto their denotations.

4.2.1 Syntax

Our language has three basic types of language constructs: programs, commands, and expressions. A program is a sequence of one or more transactions. Both single-command and multi-command transactions are allowed. Commands occur within transactions; they change relations (e.g., define a relation, modify a relation, delete a relation). Expressions occur within commands and denote a single snapshot or historical state. We represent these three types of constructs by the syntactic categories:

| | |
|-------------------|-------------------------|
| <i>PROGRAM</i> | Category of programs |
| <i>COMMAND</i> | Category of commands |
| <i>EXPRESSION</i> | Category of expressions |

We use Backus-Naur Form to specify here the syntax of programs, commands, and expressions in terms of their *immediate constituents* (i.e., the highest-level constructs that make up programs, commands, and expressions). The complete syntax of the language, including definitions of the lower-level constituents such as identifiers and snapshot states is given in Appendix C.

```

P ::= begin_transaction C commit_transaction
      | begin_transaction C abort_transaction
  
```

```

|  $P_1; P_2$ 

C ::= define_relation( $I, Y, Z$ ) | modify_relation( $I, Y', Z', E$ )
    | destroy( $I$ ) | rename_relation( $I_1, I_2$ ) |  $C_1, C_2$ 

E ::= [snapshot,  $Z, S$ ] | [historical,  $Z, H$ ] |  $I$ 
    |  $E_1 \cup E_2$  |  $E_1 - E_2$  |  $E_1 \times E_2$  |  $\pi X(E)$  |  $\sigma F(E)$ 
    |  $E_1 \dot{\cup} E_2$  |  $E_1 \dot{-} E_2$  |  $E_1 \dot{\times} E_2$  |  $\dot{\pi} X(E)$  |  $\dot{\sigma} F(E)$ 
    |  $\delta G, (I_1 := V_1, \dots, I_m := V_m)(E)$ 
    |  $\widehat{A} I_1, W, I_2, I_3, B(E_1, E_2)$  |  $\widehat{AU} I_1, W, I_2, I_3, B(E_1, E_2)$ 
    |  $\rho(I, N)$  |  $\dot{\rho}(I, N)$  |  $(E)$ 

Y' ::=  $Y$  | *

Y ::= snapshot | rollback | historical | temporal

Z' ::=  $Z$  | *

Z ::=  $(I_{1,1} : I_{1,2}, \dots, I_{m,1} : I_{m,2})$ 

```

where,

B ranges over the category $\mathcal{B}'\text{Y LIST}$;
 C, C_1 , and C_2 range over the category COMMAND ;
 E, E_1 , and E_2 range over the category EXPRESSION ;
 F ranges over the category SIGMA EXPRESSION of boolean expressions
of elements from the categories IDENTIFIER and STRING (i.e., the category
of strings in an alphabet), the relational operators, and the logical operators;
 G ranges over the category DELTA EXPRESSION of boolean expressions
of elements from the categories TIME EXPRESSION , the relational operators,
and the logical operators;
 H ranges over the category $\mathcal{H}\text{-STATE}$ of alphanumeric representations of historical
states in our historical algebra;
 $I, I_1, I_2, I_{1,1}, \dots, I_{m,2}$ range over the category IDENTIFIER of alphanumeric
identifiers;
 N ranges over the category NUMERAL of decimal numerals;
 P, P_1 , and P_2 range over the category PROGRAM ;
 S ranges over the category $\mathcal{S}\text{-STATE}$ of alphanumeric representations of snapshot
states;
 V ranges over the category TIME EXPRESSION of temporal expressions
(i.e., expressions that denote a domain of time values);
 W ranges over the category WINDOW FUNCTION of aggregation window

functions;
 X ranges over the category *IDENTIFIER LIST*;
 Y ranges over the category *CLASS* of character strings denoting relation classes; and
 Z ranges over the category *SIGNATURE* of alphanumeric representations
of signatures.

An expression, which evaluates to either a snapshot or historical state, may be a constant (i.e., an ordered triple consisting of a relation class, signature, and state); an identifier I , representing the current state of the relation denoted by I ; or an algebraic operator on either one or two other expressions. The allowable operators include the five operators that serve to define the snapshot algebra and the eight operators that serve to define our historical algebra. To these, we have added two additional operators, a rollback operator ρ and its historical counterpart $\hat{\rho}$. The rollback operator ρ takes two arguments, an identifier I and a transaction number N , and retrieves from the relation denoted by I the snapshot state current at the time of transaction N . Similarly, the rollback operator $\hat{\rho}$ retrieves from the relation denoted by I the historical state current at the time of transaction N .

EXAMPLES. The following are two examples of syntactically correct expressions in the language. The first is a constant and the second is an expression involving both a rollback operator and a constant. Their semantics will be specified in Sections 4.2.3 and 4.2.4.

```
[snapshot, (sname:string, class:string), (sname:"Phil", class:"junior"),
(sname:"Linda", class:"senior"),
(sname:"Ralph", class:"senior")]
```

```
 $\pi(\text{sname})(\rho(R1, 4)) \times [\text{snapshot}, (\text{course:string}), (\text{course}:"English")]$ 
```

Note that the alphanumeric representation of a signature includes both the names of attributes and the names of the attributes' value domains. \square

There are four commands in the language. We present here a brief description of each command, with some examples. The semantics of commands will be defined formally in Section 4.2.5.

The *define_relation* command binds a class, a signature, and an empty relation state to an identifier I .

EXAMPLE.

```
define_relation(R1, snapshot, (sname:string, class:string))
```

Here, the identifier $R1$ is defined to denote a snapshot relation with two attributes, *sname* and *class*. The contents of the relation is, by default, the empty set. \square

The `modify_relation` command may change the current class, signature, or state of a relation. Command parameters specify the new class, signature, and state. The special symbol "*" represents, depending on context, either the current class or the current signature of a relation. It may appear as a parameter in a `modify_relation` command to indicate that a relation's new class (or signature) is simply the relation's current class (signature), unchanged.

EXAMPLES.

```

modify_relation(R1, *, *, [snapshot, (sname:string, class:string),
                           (sname:"Phil", class:"junior"),
                           (sname:"Linda", class:"senior"),
                           (sname:"Ralph", class:"senior")])

modify_relation(R1, *, (sname:string, course:string),
                  π(sname)(R1) × [snapshot, (course:string),
                  (course:"English")])

modify_relation(R1, rollback, *, R1)

```

The first command changes the state of the relation denoted by `R1` but leaves the relation's class and signature unchanged. The second command changes the relation's signature and state, but not its class. The third command changes only the relation's class, as the expression `R1` evaluates to the current state of the relation. □

The `destroy` command deletes, either physically or logically, the current class, signature, and state of a relation, depending on the relation's class when the command is executed. The `rename_relation` command renames a relation by binding its current class, signature, and state to a new identifier.

EXAMPLES.

```

destroy(R1)

rename_relation(R2, R1)

```

Here we first delete the relation denoted by `R1` and then rename the relation denoted by `R2` as `R1`. □

Programs in our language contain two types of transactions, committed transactions and aborted transactions. Committed transactions are transactions, which the user initiates, that eventually commit. Aborted transactions are transactions, which the user initiates, that for some reason, dictated either by the user or by the system, abort rather than commit. The semantics of programs will be defined formally in Section 4.2.6.

4.2.2 Semantic Domains

In our language, a program denotes the database resulting from the execution of one or more transactions, in order, on an empty database. By defining the database that results from the execution of an arbitrary sequence of transactions, we specify the semantics of that transaction sequence, and hence the semantics of the language. In this section, we will define formally the flat domain (i.e., a domain with a trivial partial ordering [Schmidt 1986]) of databases; later sections will provide the connection between the syntactic category of programs and the semantic domain of databases. All domains introduced are flat domains and the notation $\{\dots\}$ is used to represent flat domains.

Assume that we are given the domain $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_e\}$, where each domain \mathcal{D}_u , $1 \leq u \leq e$, is an arbitrary, non-empty, finite or denumerable set. Also, assume that we are given the domains \mathcal{T} and $\mathcal{P}(\mathcal{T})$, where each element in \mathcal{T} represents a chronon and $\mathcal{P}(\mathcal{T})$ is the power set of \mathcal{T} . Then, we can define the following semantic domains for our language.

$$\text{TRANSACTION NUMBER} = \{0, 1, \dots\}$$

A transaction number is a non-negative integer that identifies a transaction that changes the database. The transaction number assigned to a transaction can be viewed as that transaction's time-stamp.

$$\text{RELATION CLASS} = \{\text{UNDEFINED}, \text{SNAPSHOT}, \text{ROLLBACK}, \text{HISTORICAL}, \text{TEMPORAL}\}$$

A relation is either undefined or defined to be a snapshot, rollback, historical, or temporal relation.

$$\text{RELATION SIGNATURE} = \text{IDENTIFIER} \rightarrow (\mathcal{D} + \{\text{UNBOUND}\})$$

where the notation “+” on domains means the disjoint union of domains. A relation's signature is a function that maps identifiers either onto a domain \mathcal{D}_u , $1 \leq u \leq e$ or onto UNBOUND. If a signature maps an identifier onto UNBOUND, then the identifier is unbound in that signature (i.e., it is associated with no domain). If, however, a signature maps an identifier onto a domain, then that mapping defines an attribute.

SNAPSHOT STATE = Domain of all semantically correct snapshot states (sets of m -tuples), as defined in the snapshot algebra [Maier 1983], for elements of the domain RELATION SIGNATURE and the domain $\{\mathcal{D}_1 + \dots + \mathcal{D}_e\}$, where \emptyset is the empty snapshot state. Hence, a snapshot state s on a relation signature z is a finite set of mappings from $\{I \mid z(I) \neq \text{UNBOUND}\}$ to \mathcal{D} , with the restriction that for each mapping $st \in s$, $st(I) \in z(I)$.

HISTORICAL STATE = Domain of all semantically correct historical states, as defined in our historical algebra, for elements of the domain RELATION SIGNATURE and the domain $[\{\mathcal{D}_1 + \dots + \mathcal{D}_e\} \times \mathcal{P}(\mathcal{T})]$, where \emptyset is the empty historical state.

$$\begin{aligned}
 \text{RELATION} = & [\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times \\
 & [\text{TRANSACTION NUMBER} + \{-\}]]^+ \times \\
 & [\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^* \times \\
 & [(\text{SNAPSHOT STATE} \times \text{TRANSACTION NUMBER}) + \\
 & (\text{HISTORICAL STATE} \times \text{TRANSACTION NUMBER})]^*
 \end{aligned}$$

where the special element “-” stands for the present time. A relation is thus an ordered triple consisting of

- a sequence of (relation class, transaction number, transaction number or “-”) triples,
- a sequence of (relation signature, transaction number) pairs, and
- a sequence of (relation state, transaction number) pairs.

Relations are dynamic objects whose class, signature, and state are all allowed to change over time. For example, a relation defined initially as a snapshot relation could be modified to be a historical, rollback, or temporal relation. Later, the relation could be modified to be a snapshot relation once again. Every relation always has at least one element in its class sequence, the last element recording the relation's current class (i.e., undefined, snapshot, rollback, or temporal). Any other elements in the sequence record intervals when the relation's class was either rollback or temporal.

A relation's signature (state) sequence will be empty only if the relation is currently undefined and it was never a rollback or temporal relation. If a relation is currently other than undefined, there is at least one element in its signature (state) sequence, the last element recording the relation's current signature (state). Any other elements in the sequence record the signature (state) of the relation when its class was either rollback or temporal.

The transaction-number components of all elements, but the last element, in a relation's class sequence can be viewed as time-stamps defining a fixed, closed interval during which the element's class component was the relation's class. In contrast, the third component of the last element in the sequence is always “-”; it is used to define an interval of dynamic length that always extends to the present. The transaction-number component of each element in a relation's signature (state) sequence can be viewed as a time-stamp indicating when the element's signature (state) was entered into the database and became the relation's current signature (state). Since we assume that database changes occur sequentially, the transaction-number components of a signature (state) sequence, while not necessarily consecutive, will be nevertheless strictly increasing. Thus, we can interpolate on the transaction-number component of elements in a relation's signature (state) sequence to determine the signature (state) of the relation at any time its class was either rollback or temporal.

EXAMPLE. The following is a sample relation. For notational convenience in this and later examples, we show only the attribute portion of a signature (i.e., the partial function from attribute names to value domains). Each signature maps all identifiers not shown onto UNBOUND. Also for notational convenience, we assume the natural mapping from attribute names onto attribute values for each tuple (e.g., (ename → "Phil", ssn → 250861414)).

| class | signature | state |
|-------------------------|--|---|
| ((ROLLBACK, 2, 6),) | ((((sname → string, ssn → integer), 2),)) ((sname → string, class → string), 5),) | (((), 2), ((("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)), 4), ((("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")), 5), ((({(250861414, "junior"), (147894290, "senior"), (459326889, "senior")}), 8))) |
| (SNAPSHOT, 8, -)) | ((ssn → integer, class → string), 8)) | |

The relation shown here was defined to be a rollback relation by transaction 2 and remained a rollback relation through transaction 6. While the relation was a rollback relation, all changes to its signature and state were recorded; its state was changed by transaction 4 and both its signature and its state were changed by transaction 5. Transaction 7 redefined the relation's class and the relation was last updated as a snapshot relation by transaction 8. Only when a relation's current class is either rollback or temporal is the relation treated as an append-only relation. In all other cases, updates cause outdated information to be discarded. Hence, the lack of information about the relation's class, signature, and state before transaction 2 and at transaction 7 implies that the relation was either undefined or a snapshot or historical relation at those times. Note that this relation can be rolled back only to transactions 2 through 6. Also note that the last element in the class sequence defines the relation to be a snapshot relation from transaction 8 to the present. □

DATABASE STATE = IDENTIFIER → RELATION

A database state is a function that maps each identifier onto a relation. If an identifier I is mapped onto a relation whose current class is UNDEFINED, then I denotes an undefined relation. In the empty database state, all identifiers map onto undefined relations (i.e., ((UNDEFINED, 0, -), (), ())).

DATABASE = DATABASE STATE × TRANSACTION NUMBER

A database is an ordered pair consisting of a database state and the transaction number assigned to the most recently committed transaction on the database state (i.e., the last transaction to cause a change to the database state).

4.2.3 A Semantic Type System for Expressions

Before specifying the semantics of the expressions defined syntactically in Section 4.2.1, we introduce a *semantic type system* for expressions. All syntactically correct expressions in our language are not necessarily semantically correct. An expression is semantically correct, with respect to a database state and a command, only if its evaluation on the database state during the command's execution produces either a snapshot or a historical state. Also, if the expression contains a rollback operator, it must be consistent with the class and signature of the relation being rolled back at the time of the transaction to which the relation is rolled back. Because the class and signature, as well as the state, of a relation are allowed to change over time, the semantic correctness of expressions also can vary over time. Hence, expressions that are semantically correct on a database state when one command is executed may not be semantically correct on the same database state when a subsequent command is executed (although the correctness of rollback operations to existing states will be unaffected by subsequent commands).

The semantic type system defined here allows us to do expression type-checking independent of expression evaluation. In Section 4.2.4, where we define the semantics of expressions, we will use the type system to restrict evaluation of expressions to semantically correct expressions only. Hence, any future implementation of the language can avoid the unnecessary cost associated with attempted evaluation of semantically incorrect expressions. The type system will also be used to define the semantics of commands so that commands whose execution would result in an incompatibility among a relation's class, signature, and state will never be executed. Also, separation of semantic type-checking and evaluation of expressions simplifies the formal definitions of the semantics of both expressions and commands. Note that while semantic type-checking and evaluation of some expressions (i.e., those expressions involving only constant expressions and rollback operators that roll back a relation prior to the query analysis time) can be done when a query is analyzed, most semantic type-checking and expression evaluation will have to be done when the query is executed.

Semantically correct expressions in our language evaluate to either a single snapshot state or a single historical state. We define a snapshot state's type to be an ordered pair whose first component is **SNAPSHOT** and whose second component is the state's signature. Similarly, we define a historical state's type to be an ordered pair whose first component is **HISTORICAL** and whose second component is the state's signature. A semantically correct expression's type is therefore the class and signature of the relation state resulting from the expression's evaluation and two expressions are said to be of the same type if and only if they evaluate to either snapshot or historical states on the attributes of the same signature.

We use the semantic function **T** to specify an expression's type. A semantic function is simply a function that maps a language construct onto its denotation or meaning. **T** defines an expression as a function that maps a database state and a transaction number onto either an ordered pair or **TYPEERROR**, depending on whether the expression is a semantically correct expression on the database state when a command in the transaction assigned the transaction number is executed. The ordered pair will have as its first component either

SNAPSHOT OR HISTORICAL and as its second component the signature of the relation state that the expression represents. Hence, T defines the type denotation of expressions in our language.

$$T : EXPRESSION \rightarrow [[DATABASE STATE \times TRANSACTION NUMBER] - \\ [[\{ SNAPSHOT, HISTORICAL \} \times \\ RELATION SIGNATURE] + \{ TYPEERROR \}]]$$

The result of type-checking a syntactically correct expression is the class and signature of the relation state that the expression represents if the expression is semantically correct and an error if the expression is semantically incorrect. An expression's type may depend on a database state's contents. The type of an expression involving a rollback operator also depends on the transaction number of the transaction in which the command containing the expression occurs. Hence, a database state and transaction number together define the environment in which type-checking is performed.

Before defining the semantic function T , we describe informally several functions used in its definition. Formal definitions for these auxiliary functions appear in Appendix B.

H is a semantic function that maps each alphanumeric representation of a historical state in the syntactic category $H\text{-STATE}$ onto its corresponding historical state in the semantic domain $HISTORICAL\ STATE$, if it denotes a valid historical state on a given signature. Otherwise, H maps the historical state onto $ERROR$.

N is a semantic function that maps the syntactic category $NUMERAL$ of decimal numerals into the semantic domain $INTEGER$ of integers.

S is a semantic function that maps each alphanumeric representation of a snapshot state in the syntactic category $S\text{-STATE}$ onto its corresponding snapshot state in the semantic domain $SNAPSHOT\ STATE$, if it denotes a valid snapshot state on a given signature. Otherwise, S maps the snapshot state onto $ERROR$.

$VALIDB$ is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category $BY\ LIST$ onto the boolean value $TRUE$ or $FALSE$, to indicate whether the identifiers denote a valid subset of the attributes in a given signature.

$VALIDF$ is a semantic function that maps the alphanumeric representation of a boolean predicate in the syntactic category $SIGMA\ EXPRESSION$ onto the boolean value $TRUE$ or $FALSE$, to indicate whether the predicate is a valid boolean predicate for the selection operator σ (or $\dot{\sigma}$) and a given signature.

$VALIDG$ is a semantic function that maps the alphanumeric representation of a temporal predicate in the syntactic category $DELTA\ EXPRESSION$ onto the boolean value $TRUE$ or $FALSE$, to indicate whether the predicate is a valid temporal predicate for the derivation operator δ and a given signature.

VALIDV is a semantic function that maps the alphanumeric representation of a set of assignments in the syntactic category *TIME LIST* onto the boolean value TRUE or FALSE, to indicate whether the assignments denote valid pairs of attributes and temporal expressions for the derivation operator δ and a given signature.

VALIDW is a semantic function that maps the alphanumeric representation of an aggregation windowing function in the syntactic category *WINDOW FUNCTION* onto the boolean value TRUE or FALSE, to indicate whether the function denotes a member of an arbitrary semantic domain of aggregation windowing functions.

VALIDX is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto the boolean value TRUE or FALSE, to indicate whether the identifiers denote a valid subset of the attributes in a given signature.

X is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto an element in $\wp(\text{IDENTIFIER})$, the power set of *IDENTIFIER*, if the identifiers denote a valid subset of the attributes in a given signature. Otherwise, **X** maps the list onto **ERROR**.

Y is a semantic function that maps each character string in the syntactic category *CLASS* onto the relation class that it denotes in the semantic domain *RELATION CLASS*.

Z is a semantic function that maps each alphanumeric representation of a relational signature in the syntactic category *SIGNATURE* onto its corresponding relational signature in the semantic domain *RELATION SIGNATURE*.

FindClass maps a relation onto the class component of the element in the relation's class sequence whose first transaction-number component is less than or equal to a given transaction number and whose second transaction-number component is greater than or equal to the transaction number. If no such element exists in the sequence, then *FindClass* returns **ERROR**.

FindSignature maps a relation onto the signature component of the element in the relation's signature sequence having the largest transaction-number component less than or equal to a given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindSignature* returns **ERROR**.

LastClass maps a relation onto the class component of the last element in the relation's class sequence. If the sequence is empty, *LastClass* returns **ERROR**.

LastSignature maps a relation onto the signature component of the last element in the relation's signature sequence. If the relation's signature sequence is empty, *LastSignature* returns **ERROR**.

We now define formally the semantic function **T** for each kind of expression allowed in our language. For this and later definitions of semantic functions, let e be the number of value domains D_u , $1 \leq u \leq e$, and let

d range over the domain *DATABASE STATE*.
 z , z_1 , and z_2 range over the domain *RELATION SIGNATURE*, and
 tn range over the domain *TRANSACTION NUMBER*.

```
T[[snapshot, Z, S]](d, tn) = if (Z[Z] ≠ ERROR ∧ S[S]Z[Z] ≠ ERROR)
                                then (SNAPSHOT, Z[Z])
                                else TYPEERROR
```

```
T[[historical, Z, H]](d, tn) = if (Z[Z] ≠ ERROR ∧ H[H]Z[Z] ≠ ERROR)
                                then (HISTORICAL, Z[Z])
                                else TYPEERROR
```

If a constant expression represents a snapshot state on a signature, the expression's type is the ordered pair whose first component is SNAPSHOT and whose second component is the snapshot state's signature. If a constant expression represents a historical state on a signature, the expression's type is the ordered pair whose first component is HISTORICAL and whose second component is the historical state's signature. Otherwise, evaluation of the expression's type results in an error.

EXAMPLE. For this and later examples in Section 4.2, assume that we are given the database (DS, 8) where the database state DS maps the identifier R1 onto the relation shown in the example on page 62.

```
T[[snapshot, (sname:string, class:string), (sname:"Phil", class:"junior"),
                                         (sname:"Linda", class:"senior"),
                                         (sname:"Ralph", class:"senior")]
    ]](DS, 9) = (SNAPSHOT, (sname → string, class → string))
```

Here we assume that type-checking is being performed as part of transaction 9. Note, however, that the database state is not consulted to determine the constant expression's type; the expression's type is independent of the database state. Actually, the only expressions whose type depends directly on the database state are identifiers and expressions involving the rollback operators. \square

Evaluation of a snapshot constant's type produces an error if and only if the expression does not represent a snapshot state on a signature. As we will see in Section 4.2.4, evaluation of a constant expression's type produces an error under exactly the same conditions that evaluation of the expression produces an error. This relationship between a constant expression's type and value is both a necessary and a sufficient condition to ensure that the evaluation of any expression will result in an error when evaluation of the expression's type results in an error.

```

 $\mathbf{T}[I](d, tn) = \begin{array}{ll} \text{if } & (\text{LastClass}(d(I)) = \text{SNAPSHOT} \\ & \vee \text{LastClass}(d(I)) = \text{ROLLBACK}) \\ \text{then } & (\text{SNAPSHOT}, \text{LastSignature}(d(I))) \\ \text{else if } & (\text{LastClass}(d(I)) = \text{HISTORICAL} \\ & \vee \text{LastClass}(d(I)) = \text{TEMPORAL}) \\ \text{then } & (\text{HISTORICAL}, \text{LastSignature}(d(I))) \\ \text{else } & \text{TYPEERROR} \end{array}$ 

```

where the notation $d(I)$ stands for the relation denoted by the identifier I in the database state d . The type of an expression I is the ordered pair whose first component is **SNAPSHOT** if I 's current class is either snapshot or rollback and **HISTORICAL** if its current class is either historical or temporal. The ordered pair's second component is always I 's current signature. An error occurs if the relation is currently undefined.

EXAMPLE.

$\mathbf{T}[R1](DS, 9) = (\text{SNAPSHOT}, (\text{ssn} \rightarrow \text{integer}, \text{class} \rightarrow \text{string}))$

□

$\mathbf{T}[E_1 \cup E_2](d, tn) = \begin{array}{ll} \text{if } & \mathbf{T}[E_1](d, tn) = \mathbf{T}[E_2](d, tn) = (\text{SNAPSHOT}, z) \\ \text{then } & \mathbf{T}[E_1](d, tn) \\ \text{else } & \text{TYPEERROR} \end{array}$

$\mathbf{T}[E_1 - E_2](d, tn) = \begin{array}{ll} \text{if } & \mathbf{T}[E_1](d, tn) = \mathbf{T}[E_2](d, tn) = (\text{SNAPSHOT}, z) \\ \text{then } & \mathbf{T}[E_1](d, tn) \\ \text{else } & \text{TYPEERROR} \end{array}$

```

T[E1 × E2](d, tn) =
  if   (T[E1](d, tn) = (SNAPSHOT, z1) ∧ T[E2](d, tn) = (SNAPSHOT, z2)
        ∧ ∀I, I ∈ IDENTIFIER, (z1(I) = UNBOUND ∨ z2(I) = UNBOUND))
  then (SNAPSHOT, {(I, Du) | 1 ≤ u ≤ e ∧ ((I, Du) ∈ z1 ∨ (I, Du) ∈ z2)}
        ∪ {(I, UNBOUND) | I ∈ IDENTIFIER ∧ (I, UNBOUND) ∈ z1
        ∨ (I, UNBOUND) ∈ z2})
  else TYPEERROR

```

```

T[π X(E)](d, tn) =
  if   (T[E](d, tn) = (SNAPSHOT, z) ∧ VALIDX[X] z)
  then (SNAPSHOT, {(I, Du) | I ∈ X[X] z ∧ 1 ≤ u ≤ e ∧ (I, Du) ∈ z}
        ∪ {(I, UNBOUND) | I ∉ X[X] z ∧ I ∈ IDENTIFIER})
  else TYPEERROR

```

```

T[σ F(E)](d, tn) = if   (T[E](d, tn) = (SNAPSHOT, z) ∧ VALIDF[F] z)
  then T[E](d, tn)
  else TYPEERROR

```

The type of an expression involving one of the five basic snapshot operators is an ordered pair whose first component is SNAPSHOT and whose second component is the signature of the relation state produced when the expression is evaluated, if two conditions are met. The first component of the type of all subexpressions must be SNAPSHOT and the second component of the type of all subexpressions must be a signature satisfying any restrictions placed on the signatures of relation states in corresponding expressions in the snapshot algebra. For example, our definitions of union and difference require that the signatures for E₁ and E₂ be identical while our definition of cartesian product requires that the attributes defined by the signatures for E₁ and E₂ be disjoint. (Note that we can eliminate this last restriction and effectively allow the cartesian product of snapshot states on arbitrary signatures through the introduction of a simple attribute renaming operator [Maier 1983] into the language.) If either condition is not met, evaluation of the expression's type results in an error.

```

 $T[\rho(I, N)](d, tn) = \begin{cases} \text{if } N[N] < tn \wedge FindClass(d(I), N[N]) = \text{ROLLBACK} \\ \text{then } (\text{SNAPSHOT}, FindSignature(d(I), N[N])) \\ \text{else } \text{TYPEERROR} \end{cases}$ 

```

A rollback expression's type is the ordered pair whose first component is **SNAPSHOT** and whose second component is the signature of the relation denoted by I when transaction $N[N]$ was processed, if the relation was a rollback relation at that time. Otherwise, evaluation of the expression's type results in an error. Because we assume sequential transaction processing, tn is the transaction number of the one active transaction and all transactions with a transaction number less than tn are committed. Hence, we allow rollback only to committed transactions.

EXAMPLES.

$$T[\rho(R1, 4)](DS, 9) = (\text{SNAPSHOT}, (\text{surname} \rightarrow \text{string}, \text{ssn} \rightarrow \text{integer}))$$

$$T[\pi(\text{surname})(\rho(R1, 4))](DS, 9) = (\text{SNAPSHOT}, (\text{surname} \rightarrow \text{string}))$$

$$T[\pi(\text{surname})(\rho(R1, 4)) \times [\text{snapshot}, (\text{course:string}), (\text{course:"English"})]](DS, 9) = (\text{SNAPSHOT}, (\text{surname} \rightarrow \text{string}, \text{course} \rightarrow \text{string}))$$

□

We now present the definitions of the semantic function **T** for expressions involving historical operators. The type denotation of an expression involving a historical operator is defined identically to that of an expression involving an analogous snapshot operator (if one exists), with the exception that **HISTORICAL** and **TEMPORAL** are substituted for **SNAPSHOT** and **ROLLBACK**, respectively.

$$\begin{aligned}
 T[E_1 \cup E_2](d, tn) = & \text{ if } T[E_1](d, tn) = T[E_2](d, tn) = (\text{HISTORICAL}, z) \\
 & \text{then } T[E_1](d, tn) \\
 & \text{else } \text{TYPEERROR}
 \end{aligned}$$

$$\begin{aligned}
 T[E_1 - E_2](d, tn) = & \text{ if } T[E_1](d, tn) = T[E_2](d, tn) = (\text{HISTORICAL}, z) \\
 & \text{then } T[E_1](d, tn) \\
 & \text{else } \text{TYPEERROR}
 \end{aligned}$$

$\mathbf{T}[\![E_1 \times E_2]\!](d, tn) =$

if $(\mathbf{T}[\!E_1]\!)(d, tn) = (\text{HISTORICAL}, z_1) \wedge \mathbf{T}[\!E_2]\!)(d, tn) = (\text{HISTORICAL}, z_2)$
 $\wedge \forall I, I \in \text{IDENTIFIER}, (z_1(I) = \text{UNBOUND} \vee z_2(I) = \text{UNBOUND}))$

then $(\text{HISTORICAL}, \{(I, D_u) \mid 1 \leq u \leq e \wedge ((I, D_u) \in z_1 \vee (I, D_u) \in z_2)\})$
 $\cup \{(I, \text{UNBOUND}) \mid I \in \text{IDENTIFIER} \wedge (I, \text{UNBOUND}) \in z_1$
 $\wedge (I, \text{UNBOUND}) \in z_2\})$

else TYPEERROR

$\mathbf{T}[\!\hat{\pi} X(E)]\!(d, tn) =$

if $(\mathbf{T}[\!E]\!)(d, tn) = (\text{HISTORICAL}, z) \wedge \text{VALIDX}[X]z$

then $(\text{HISTORICAL}, \{(I, D_u) \mid I \in \mathbf{X}[X]z \wedge 1 \leq u \leq e \wedge (I, D_u) \in z\})$
 $\cup \{(I, \text{UNBOUND}) \mid I \notin \mathbf{X}[X]z \wedge I \in \text{IDENTIFIER}\})$

else TYPEERROR

$\mathbf{T}[\!\hat{\sigma} F(E)]\!(d, tn) =$ if $(\mathbf{T}[\!E]\!)(d, tn) = (\text{HISTORICAL}, z) \wedge \text{VALIDF}[F]z$
 $\text{then } \mathbf{T}[\!E]\!(d, tn)$
 $\text{else } \text{TYPEERROR}$

$\mathbf{T}[\!\delta G, (I_1 := V_1, \dots, I_m := V_m)(E)]\!(d, tn) =$

if $(\mathbf{T}[\!E]\!)(d, tn) = (\text{HISTORICAL}, z)$
 $\wedge \text{VALIDG}[G]z \wedge \text{VALIDV}[(I_1 := V_1, \dots, I_m := V_m)]z$

then $\mathbf{T}[\!E]\!(d, tn)$
 $\text{else } \text{TYPEERROR}$

```

T[ $\widehat{A} I_1, W, I_2, I_3, B(E_1, E_2)$ ](d, tn) =
  if   (T[E1](d, tn) = (HISTORICAL, z1)  $\wedge$  T[E2](d, tn) = (HISTORICAL, z2)
         $\wedge$  z1  $\subseteq$  z2  $\wedge$  VALIDB[B]z1  $\wedge$  I2  $\in$  z1  $\wedge$  I3  $\notin$  z1
         $\wedge$  VALIDSA[I1]  $\neq$  ERROR  $\wedge$  VALIDW[W]  $\neq$  ERROR)
    then (HISTORICAL, B[B]z1  $\cup$  {I3})
  else TYPEERROR

```

where, we assume that VALIDSA is a semantic function that determines whether an identifier maps onto the name of an aggregate family in an arbitrary domain of scalar aggregate families.

```

T[ $\widehat{AU} I_1, W, I_2, I_3, B(E_1, E_2)$ ](d, tn) =
  if   (T[E1](d, tn) = (HISTORICAL, z1)  $\wedge$  T[E2](d, tn) = (HISTORICAL, z2)
         $\wedge$  z1  $\subseteq$  z2  $\wedge$  VALIDB[B]z1  $\wedge$  I2  $\in$  z1  $\wedge$  I3  $\notin$  z1
         $\wedge$  VALIDSA[I1]  $\neq$  ERROR  $\wedge$  VALIDW[W]  $\neq$  ERROR)
    then (HISTORICAL, B[B]z1  $\cup$  {I3})
  else TYPEERROR

```

```

T[ $\rho(I, N)$ ](d, tn) = if   N[N] < tn  $\wedge$  FindClass(d(I), N[N]) = TEMPORAL
                           then (HISTORICAL, FindSignature(d(I), N[N]))
                         else TYPEERROR

```

Finally, we present the definition of the semantic function T for the last expression construct, which is used to group subexpressions.

```
T[(E)](d, tn) = T[E](d, tn)
```

4.2.4 Expressions

The semantic function E defines the denotation of expressions in our language. E defines an expression as a function that maps a database state and a transaction number onto either a snapshot state (i.e. an element of the *SNAPSHOT STATE* semantic domain), a historical state (i.e., an element of the *HISTORICAL STATE* semantic domain), or ERROR.

$$E : EXPRESSION \rightarrow [[DATABASE STATE \times TRANSACTION NUMBER] \rightarrow [SNAPSHOT STATE + HISTORICAL STATE + \{ ERROR \}]]$$

If an expression is a semantically correct expression on a database state, expression evaluation on the database state produces either a snapshot state or a historical state. Otherwise, expression evaluation produces an error. The environment for expression evaluation, a database state and the transaction number of the active transaction, is the same as that for expression type-checking. Note that expression evaluation has no side-effect; it leaves the database state unchanged.

Before defining the semantic function E, we describe informally additional auxiliary functions used in E's definition. Formal definitions for these functions appear in Appendix B.

B is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *BY LIST* onto an element in $\wp(Identifier)$, the power set of Identifier, if the identifiers denote a valid subset of the attributes in a given signature. Otherwise, B maps the list onto ERROR.

F is a semantic function that maps the alphanumeric representation of a boolean predicate in the syntactic category *SIGMA EXPRESSION* onto its corresponding boolean predicate in the semantic domain *SELECTION PREDICATE*, if it denotes a valid boolean predicate for the selection operator σ (or δ) and a given signature. Otherwise, F maps the expression onto ERROR.

G is a semantic function that maps the alphanumeric representation of a temporal predicate in the syntactic category *DELTA EXPRESSION* onto its corresponding temporal predicate in the semantic domain *DERIVATION PREDICATE*, if it denotes a valid temporal predicate for the derivation operator δ and a given signature. Otherwise, G maps the expression onto ERROR.

V is a semantic function that maps the alphanumeric representation of a set of assignments in the syntactic category *TIME LIST* onto its corresponding set of ordered pairs in the semantic domain $\wp(Identifier \times TEMPORAL EXPRESSION)$, if all the assignments denote valid pairs of attributes and temporal expressions for the derivation operator δ and a given signature. Otherwise, V maps the assignment onto ERROR.

W is a semantic function that maps the alphanumeric representation of an aggregation windowing function in the syntactic category *WINDOW FUNCTION* onto an element in the arbitrary semantic domain *AGGREGATION WINDOW FUNCTION*, if the function denotes a member of this semantic domain. Otherwise, W maps the function onto ERROR.

FindState maps a relation onto the state component of the element in the relation's state sequence having the largest transaction-number component less than or equal to a

given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindState* returns ERROR.

LastState maps a relation onto the state component of the last element in the relation's state sequence. If the relation's state sequence is empty, *LastState* returns ERROR.

We now define formally the semantic function E for each kind of expression allowed in the language.

$$\begin{aligned} E[[\text{snapshot}, Z, S]](d, tn) = & \text{ if } T[[\text{snapshot}, Z, S]](d, tn) \neq \text{TYPEERROR} \\ & \text{then } S[S]Z[Z] \\ & \text{else } \text{ERROR} \end{aligned}$$

$$\begin{aligned} E[[\text{historical}, Z, H]](d, tn) = & \text{ if } T[[\text{historical}, Z, H]](d, tn) \neq \text{TYPEERROR} \\ & \text{then } H[H]Z[Z] \\ & \text{else } \text{ERROR} \end{aligned}$$

EXAMPLE.

$$\begin{aligned} E[[\text{snapshot}, (\text{sname:string, class:string}), (\text{sname:"Phil", class:"junior"}), \\ (\text{sname:"Linda", class:"senior"}), (\text{sname:"Ralph", class:"senior"})]](\text{DS.9}) = & \{ ("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior") \} \end{aligned}$$

□

$$E[I](d, tn) = \text{ if } T[I](d, tn) \neq \text{TYPEERROR} \text{ then } \text{LastState}(d(I)) \text{ else } \text{ERROR}$$

An identifier expression, if semantically correct, always evaluates to the current state of the relation denoted by I.

EXAMPLE.

$$E[R1](\text{DS.9}) = \{ (250861414, "junior"), (147894290, "senior"), (459326889, "senior") \}$$

□

$E[E_1 \cup E_2](d, tn) = \begin{cases} \text{if } T[E_1 \cup E_2](d, tn) \neq \text{TYPEERROR} \\ \text{then } E[E_1](d, tn) \cup E[E_2](d, tn) \\ \text{else } \text{ERROR} \end{cases}$

$E[E_1 - E_2](d, tn) = \begin{cases} \text{if } T[E_1 - E_2](d, tn) \neq \text{TYPEERROR} \\ \text{then } E[E_1](d, tn) - E[E_2](d, tn) \\ \text{else } \text{ERROR} \end{cases}$

$E[E_1 \times E_2](d, tn) = \begin{cases} \text{if } T[E_1 \times E_2](d, tn) \neq \text{TYPEERROR} \\ \text{then } E[E_1](d, tn) \times E[E_2](d, tn) \\ \text{else } \text{ERROR} \end{cases}$

$E[\pi X(E)](d, tn) = \begin{cases} \text{if } T[\pi X(E)](d, tn) = (\text{SNAPSHOT}, z) \\ \text{then } \pi_{X[X]}(E[E](d, tn)) \\ \text{else } \text{ERROR} \end{cases}$

$E[\sigma F(E)](d, tn) = \begin{cases} \text{if } T[\sigma F(E)](d, tn) = (\text{SNAPSHOT}, z) \\ \text{then } \sigma_{F[F]}(E[E](d, tn)) \\ \text{else } \text{ERROR} \end{cases}$

For each of the five snapshot operators, the denotation of a semantically correct expression containing the operator is defined as the standard snapshot operator over the denotation of the argument(s) to that operator.

$E[\rho(I, N)](d, tn) = \begin{cases} \text{if } T[\rho(I, N)](d, tn) \neq \text{TYPEERROR} \\ \text{then } \text{FindState}(d(I), N[N]) \\ \text{else } \text{ERROR} \end{cases}$

A semantically correct rollback expression evaluates to the snapshot state of the relation denoted by I at the time of transaction $N[N]$. The rollback operator always rolls a relation backward, but never forward, in time. Because transactions always update the database as they are executed, it is impossible to roll a relation forward in time. Although

relations can't be rolled forward in time, our orthogonal treatment of valid and transaction time provides support for both *retroactive* changes and *postactive* changes (i.e., changes that will occur in the future) [Snodgrass & Ahn 1985]. Recall from the definition of the semantic function T that a rollback expression is semantically correct only if the relation was a rollback relation when the transaction was processed.

EXAMPLES.

$$E[\rho(R1, 4)](DS, 9) = \{ ("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889) \}$$

$$E[\pi(sname)(\rho(R1, 4))](DS, 9) = \{ ("Phil"), ("Linda"), ("Ralph") \}$$

$$E[\pi(sname)(\rho(R1, 4)) \times [\text{snapshot}, (\text{course:string}), (\text{course:"English"})]](DS, 9) = \{ ("Phil", "English"), ("Linda", "English"), ("Ralph", "English") \}$$

□

We now present the definitions of the semantic function E for expressions involving historical operators. The denotation of an expression involving a historical operator is defined identically to that of an expression involving an analogous snapshot operator (if one exists).

$$E[E_1 \dot{\cup} E_2](d, tn) = \begin{array}{ll} \text{if } & T[E_1 \dot{\cup} E_2](d, tn) \neq \text{TYPEERROR} \\ & \text{indexhistorical operators!union} \\ \text{then } & E[E_1](d, tn) \dot{\cup} E[E_2](d, tn) \\ \text{else } & \text{ERROR} \end{array}$$

$$E[E_1 \dot{-} E_2](d, tn) = \begin{array}{ll} \text{if } & T[E_1 \dot{-} E_2](d, tn) \neq \text{TYPEERROR} \\ \text{then } & E[E_1](d, tn) \dot{-} E[E_2](d, tn) \\ \text{else } & \text{ERROR} \end{array}$$

$$E[E_1 \dot{\times} E_2](d, tn) = \begin{array}{ll} \text{if } & T[E_1 \dot{\times} E_2](d, tn) \neq \text{TYPEERROR} \\ \text{then } & E[E_1](d, tn) \dot{\times} E[E_2](d, tn) \\ \text{else } & \text{ERROR} \end{array}$$

$E[\hat{\pi} X(E)](d, tn) = \text{if } T[\hat{\pi} X(E)](d, tn) = (\text{HISTORICAL}, z)$
 then $\hat{\pi}_{X[X]}(E[E](d, tn))$
 else ERROR

$E[\hat{\sigma} F(E)](d, tn) = \text{if } T[\hat{\sigma} F(E)](d, tn) = (\text{HISTORICAL}, z)$
 then $\hat{\sigma}_{F[F]}(E[E](d, tn))$
 else ERROR

$E[\delta G, (I_1 := V_1, \dots, I_m := V_m)(E)](d, tn) =$
 if $T[\delta G, (I_1 := V_1, \dots, I_m := V_m)(E)](d, tn) = (\text{HISTORICAL}, z)$
 then $\delta_{G[G]}(V[(I_1 := V_1, \dots, I_m := V_m)]_z(E[E](d, tn)))$
 else ERROR

$E[\hat{A} I_1, W, I_2, I_3, B(E_1, E_2)](d, tn) =$
 if $T[\hat{A} I_1, W, I_2, I_3, B(E_1, E_2)](d, tn) = (\text{HISTORICAL}, z)$
 then $\hat{A}_{SA[I_1], W[W], I_2, I_3, B[B]}(E[E_1](d, tn), E[E_2](d, tn))$
 else ERROR

where, we assume **SA** is a semantic function that maps identifiers onto the aggregate family that they name in an arbitrary domain of scalar aggregate families.

$E[\widehat{AU} I_1, W, I_2, I_3, B(E_1, E_2)](d, tn) =$
 if $T[\widehat{AU} I_1, W, I_2, I_3, B(E_1, E_2)](d, tn) = (\text{HISTORICAL}, z)$
 then $\widehat{AU}_{SA[I_1], W[W], I_2, I_3, B[B]}(E[E_1](d, tn), E[E_2](d, tn))$
 else ERROR

$E[\hat{\rho}(I, N)](d, tn) = \text{if } T[\hat{\rho}(I, N)](d, tn) \neq \text{TYPEERROR}$
 then $FindState(d(I), N[N])$
 else ERROR

We now present the definition of the semantic function **E** for the expression construct

that groups subexpressions.

$$\mathbf{E}[(E)](d, tn) = \mathbf{E}[E](d, tn)$$

4.2.5 Commands

The semantic function \mathbf{C} defines the denotation of commands defined syntactically in Section 4.2.1. \mathbf{C} defines a command as a function that maps a database state and a transaction number onto a database state and a status code. Execution of a semantically correct command produces a new database state and the status code **OK**, indicating that the command was successfully executed. Execution of a semantically incorrect command produces the original database state unchanged and the status code **ERROR**, indicating that the command could not be executed.

$$\begin{aligned} \mathbf{C} : \text{COMMAND} &\rightarrow [(\text{DATABASE STATE} \times \text{TRANSACTION NUMBER}) - \\ &[\text{DATABASE STATE} \times \{\text{OK}, \text{ERROR}\}]] \end{aligned}$$

The environment for command execution is the same as that for expression type-checking and evaluation, a database state and the transaction number of the active transaction (i.e., the transaction in which the command being executed occurs). A command produces a new database state from the given database state by changing a relation.

We use semantic type-checking of expressions in the definition of \mathbf{C} to restrict evaluation of expressions to semantically correct expressions only. We also incorporate error-checking, based on the type system for expressions, into \mathbf{C} 's definition to guarantee consistency among a relation's class, signature, and state following update. Error-checking ensures that commands actually change relations only when the change would result in a relation with compatible class, signature, and state. Commands whose execution would result in an inconsistency among a relation's class, signature, and state are effectively ignored (i.e., they do not alter the database state).

Before defining the semantic function \mathbf{C} , we describe informally several functions used in its definition. Formal definitions for these functions appear in Appendix B.

\mathbf{Y}' is the same as the semantic function \mathbf{Y} with the exception that it maps the special symbol $*$ onto a relation's current class.

\mathbf{Z}' is the same as the semantic function \mathbf{Z} with the exception that it maps the special symbol $*$ onto a relation's current signature.

Consistent is a boolean function that determines whether a class and signature are consistent with an expression's type.

MSoT (*Modified Start of Transaction*) is a function that maps a relation and a transaction number onto the history of the relation as a rollback or temporal relation prior to the

start of the transaction assigned the transaction number. We refer to this history as the relation's MSoT for that transaction. The significance of *MSoT* will become apparent when we discuss multiple-command transactions.

EXAMPLE. Again assume, as in earlier examples, that we are given the database (DS, 8) where the database state component maps the identifier R1 onto the relation shown in the example on page 62.

$MSoT(R1, 9) =$

| class | signature | state |
|--------------------|--|--|
| ((ROLLBACK, 2, 6)) | ((((sname → string, ssn → integer), 2), ((sname → integer, class → string), 5))) | (((), 2), (({("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 4), (({("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")}, 5))) |

In this example, *MSoT* retains R1's history as a rollback relation prior to transaction 9. Although R1's current class, signature, and state were recorded before the start of transaction 9, they have been discarded because they are not part of R1's history as a rollback relation. If, however, the last element in R1's class sequence had been (ROLLBACK, 8, -), then R1's current class, signature, and state also would have been retained. In this case, *MSoT* simply would have changed the second transaction-number component of the last element in R1's class sequence to 8 to indicate that the resulting relation only records R1's history as a rollback relation through transaction 8. If R1 had never been a rollback or temporal relation, then *MSoT* would have mapped R1 onto ((), (), ()). □

Expand replaces the second transaction-number component in the last element of a relation's MSoT class sequence with the special element "-". *Expand* has the effect of making the length of the interval for the class component of this element dynamic, extending to the present.

NewSignature maps a relation's MSoT and a (signature, transaction number) pair onto the empty sequence, if the signature in the last element of the relation's MSoT signature sequence is equal to the signature in the (signature, transaction number) pair, or a one-element sequence containing the (signature, transaction number) pair, otherwise.

NewState maps a relation's MSoT, a (relation state, transaction number) pair, and a (class, signature) pair onto the empty sequence, if the class and signature in the last elements of the relation's MSoT class and signature sequences are consistent with the (class, signature) pair and the state in the last element of the relation's MSoT state sequence

is equal to the relation state in the (relation state, transaction number) pair, or a one-element sequence containing the (relation state, transaction number) pair, otherwise.

We define formally the semantics of commands using the same approach we used to define the semantics of expressions. We define the semantic function C for each kind of command allowed in the language. In each of the following definitions, the predicate specifies the conditions under which the command is executed. If these conditions hold, a new database state is produced and the status code `OK` is returned; otherwise, the database state is left unchanged and the status code `ERROR` is returned. The conditions specified in each definition are both necessary and sufficient to ensure that only semantically correct expressions are evaluated and that the class, signature, and state of each relation in the database state following execution of the command are consistent. In all five definitions we assume that if a_1, a_2, a_3, b_1, b_2 , and b_3 are all sequences, then $(a_1, a_2, a_3) \parallel_3 (b_1, b_2, b_3)$ denotes the triple $(a_1 \parallel b_1, a_2 \parallel b_2, a_3 \parallel b_3)$, where " \parallel " is the concatenation operator on sequences. Also, the notation $d[r/I]$ stands for a new database state that differs from the database state d only in that it maps the identifier I onto the relation r .

Defining a Relation

The `define_relation` command assigns to a relation, whose current class is `UNDEFINED`, a new class and signature and the empty relation state consistent with the new class. The assignment becomes effective when the transaction in which the command occurs is committed. The changes that the command makes to the relation to effect this assignment depend on the relation's current class; the last class, signature, and state, if any, in the relation's MSoT for the transaction in which the command occurs; and whether the new class is a single-state class (i.e., `SNAPSHOT` or `HISTORICAL`) or a multi-state class (i.e., `ROLLBACK` or `TEMPORAL`). We hereafter refer to the last class, signature, and state in a relation's MSoT, if present, as the relation's MSoT class, signature, and state, respectively. The actions performed by the `define_relation` command, for all possible combinations of these variables, can be reduced to the three cases shown in Table 4.1.

If the relation's current class is `UNDEFINED`, the `define_relation` command replaces the relation with its MSoT, augmented to include the new class, signature, and state. If the new class represents a non-disjoint extension of the relation's MSoT class, the interval assigned the MSoT class is extended (i.e., made into a dynamically expanding interval by changing the second transaction-number component to "-") to include the transaction in which the command occurs. This case is limited to `define_relation` commands in multiple-command transactions, which we discuss at the end of this section. Otherwise, the new class is appended to the MSoT class sequence. In either case, a new signature (state) is added to the MSoT signature (state) sequence only if it differs from the MSoT signature (state). If the relation's current class is other than `UNDEFINED`, the command encounters an error condition and leaves the relation unchanged.

The formal definition of `define_relation` follows directly from Table 4.1.

| <i>Current Class</i> | | <i>New Class</i> | |
|----------------------|--|--|---|
| | | SingleStateClass | MultiStateClass |
| | New Class Extends MSoT Class | Not Applicable | ¹ Extend MSoT Append to MSoT, if Changed Append to MSoT, if Changed |
| Undefined | New Class Does Not Extend MSoT Class | ² Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed | Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed |
| SingleStateClass | | ³ Error | Error |
| MultiStateClass | | Error | Error |

Table 4.1: Define Relation Command

```

C[define_relation(I, Y, Z)](d, tn) =
  if      (M = MSoT(d(I), tn)  $\wedge$  LastClass(d(I)) = UNDEFINED
            $\wedge$  Y[Y]  $\neq$  ERROR  $\wedge$  Z[Z]  $\neq$  ERROR)
  then if    FindClass(M, tn - 1) = Y[Y]
  then (d[(Expand(M) ||3 (( ), NewSignature(M, (Z[Z], tn)),
                           NewState(M, (Ø, tn), (Y[Y], Z[Z]))))
           )/I], ok)
  else  (d[(M ||3 ((Y[Y], tn, -)), NewSignature(M, (Z[Z], tn)),
                           NewState(M, (Ø, tn), (Y[Y], Z[Z)))))
           )/I], ok)
  else  (d, ERROR)

```

where M ranges over the domain $RELATION + \{((), (), ())\}$.

EXAMPLES. In these, and later examples, we show the result of executing a sequence of commands, starting with the database (DS. 8). We assume that each command corresponds to a single-command transaction that commits. For simplicity, we always refer to the current database state as DS, although it changes with each command's execution (i.e., transaction's commitment). We also restrict the commands to the relations denoted by the identifiers R1, R2, and R3 and show only the portion of the database state changed by each command's execution. We assume that DS maps the identifiers R2 and R3 onto the following relations.

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|-----|--|--|--|
| R2→ | $\langle(\text{ROLLBACK}, 1, 5),$ $(\text{UNDEFINED}, 6, -)\rangle$ | $\langle((\text{ename} \rightarrow \text{string},$ $\text{ssn} \rightarrow \text{integer}), 1)$ () | $\langle(\emptyset, 1),$ $(\{\text{"Phil"}, 250861414),$ $(\text{"Linda"}, 147894290),$ $(\text{"Ralph"}, 459326889)\}, 3)$ () |
| R3→ | $\langle(\text{UNDEFINED}, 0, -)\rangle$ | $\langle\rangle$ | $\langle\rangle$ |

Note that a relation whose current class is UNDEFINED has neither a current signature nor a current state. The relation denoted by R2 has a MSoT signature (state), but not a current signature (state). The relation denoted by R3 has neither a MSoT signature (state) nor a current signature (state).

C[define_relation(R2, rollback, (ename:string, ssn:integer))](DS. 9)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|-----|---|--|--|
| R2→ | $\langle(\text{ROLLBACK}, 1, 5),$ $(\text{ROLLBACK}, 9, -)\rangle$ | $\langle((\text{ename} \rightarrow \text{string},$ $\text{ssn} \rightarrow \text{integer}), 1)$ () | $\langle(\emptyset, 1),$ $(\{\text{"Phil"}, 250861414),$ $(\text{"Linda"}, 147894290),$ $(\text{"Ralph"}, 459326889)\}, 3)$ $(\emptyset, 9)$ () |

C[**define_relation(R3, snapshot, (sname:string, class:string))**](DS, 10)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|-------------------------|--|---------------|
| R3 → | ((SNAPSHOT, 10, -)) | ((sname → string, class → string), 10)) | ((Ø, 10)) |

The first command makes the relation denoted by R2 a rollback relation over the attributes *sname* and *ssn*, effective when transaction 9 commits. Although the new class and the relation's MSoT class are equal, the intervals associated with the two are disjoint. Hence, the new class is appended to the relation's MSoT class sequence. The new signature is not appended to the relation's MSoT signature sequence because it is the same as the relation's MSoT signature. The new state, the empty set, differs from the relation's MSoT state. Hence, it is added to the relation's MSoT state sequence. The second command makes the relation denoted by R3 a snapshot relation over the attributes *sname* and *class*, effective when transaction 10 commits. Because the relation's MSoT at transaction 10 is ((), (), ()), the command transforms the relation's class, signature, and state sequences into single-element sequences containing the new class, signature, and state. Note that information about both relations when they were undefined has been discarded as it is not needed for rollback. □

Modifying a Relation

The **modify_relation** command assigns to a relation, whose current class is other than **UNDEFINED**, a new class, signature, and relation state. The assignment becomes effective when the transaction in which the command occurs is committed. The **modify_relation** command differs from the **define_relation** command in only three respects. First, the **modify_relation** command only updates a relation if its current class is not **UNDEFINED**, whereas the **define_relation** command does just the opposite. Second, the **modify_relation** command, unlike the **define_relation** command, allows the new class (signature) to be the relation's current class (signature). Third, the **modify_relation** command allows the new relation state to be the value of any semantically correct expression consistent with the new class and signature, whereas the **define_relation** command requires that the new state be the empty state consistent with the new class. Otherwise, the semantics of the two commands is the same. The actions performed by the **define_relation** command are summarized in Table 4.2.

The formal definition of **modify_relation** follows directly from the above description of the command and Table 4.2.

| <i>Current Class</i> | <i>New Class</i> | |
|---|---|--|
| | <i>SingleStateClass</i> | <i>MultiStateClass</i> |
| <i>SingleStateClass</i> or <i>MultiStateClass</i> | <i>New Class Extends MSoT Class</i> Not Applicable | 1 Extend MSoT Append to MSoT, if Changed Append to MSoT, if Changed |
| <i>SingleStateClass</i> or <i>MultiStateClass</i> | <i>New Class Does Not Extend MSoT Class</i> 2 Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed | 2 Append to MSoT Append to MSoT, if Changed Append to MSoT, if Changed |
| <i>Undefined</i> | 3 Error | Error |

Table 4.2: Modify Relation Command

```

C[modify_relation(I, Y', Z', E)](d, tn) =
  if      (M = MSoT(d(I), tn) ∧ T[E](d, tn) ≠ ERROR ∧ LastClass(d(I)) ≠ UNDEFINED
            ∧ Consistent(Y'[Y'](d(I)), Z'[Z'](d(I)), T[E](d, tn)))
    then if      FindClass(M, tn - 1) = Y'[Y'](d(I))
      then (d[(Expand(M) ||3 (( ), NewSignature(M, (Z'[Z'](d(I)), tn)),
                           NewState(M, (E[E](d, tn), tn), T[E](d, tn)))
                           )/I], ok)
    else (d[(M ||3 (((Y'[Y'](d(I)), tn, -)), NewSignature(M, (Z'[Z'](d(I)), tn)),
                           NewState(M, (E[E](d, tn), tn), T[E](d, tn)))
                           )/I], ok)
  else (d, ERROR)

```

If a relation's current class is other than UNDEFINED, the `modify_relation` command replaces the relation with its MSoT, augmented to include the new class, signature, and state. If the relation's current class is UNDEFINED, the command encounters an error and leaves the relation unchanged.

EXAMPLES.

C[modify_relation(R2, *, *, $\rho(R2,5) - \sigma$ ename="Ralph"($\rho(R2,5)$))](DS. 11)

C[modify_relation(R3, *, *, ρ(R1,5))] (DS, 12)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|---------------------------|---|--|
| R3 → | ((SNAPSHOT, 12, --))) | ((((sname → string, class → string), 12))) | (({{("Phil", "junior"), ("Linda", "senior"), ("Ralph", "senior")}}, 12)) |

The first command changes the state of the relation denoted by R2 while the second command changes the state of the relation denoted by R3. The commands, however, do not change the class or signature of either relation. For the first command, the new class (i.e., R2's current class) is a non-disjoint extension of R2's MSoT class. Hence, the interval for R2's MSoT class is made into a dynamically expanding interval that includes transaction 11, but no new element is added to R2's MSoT class sequence. The new signature (i.e., R2's current signature) is the same as R2's MSoT signature, hence it is not added to R2's MSoT signature sequence. The new state differs from R2's MSoT state, hence it is appended to R2's MSoT state sequence. Because R3's MSoT at transaction 12 is still ((), (), ()), the second command transforms R3's class, signature, and state sequences into single-element sequences containing the new class (i.e., R3's current class), signature (i.e., R3's current signature), and state. Note that R2's state at transaction 9 through transaction 10 has been retained and remains accessible via the rollback operator ρ , but R3's state before transaction 12 has been discarded (i.e., physically deleted from the database state).

C[**modify_relation(R3, *, (sname:string, course:string),**
 $\pi(\text{sname})(\text{R3}) \times [\text{snapshot, (course:string)},$
 $(\text{course}:\text{"English"})]]$] (DS, 13)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|--|--|--|
| R3 → | $\langle (\text{SNAPSHOT}, 13, -) \rangle$ | $\langle ((\text{sname} \rightarrow \text{string},$ $\text{course} \rightarrow \text{string}), 13) \rangle$ | $\langle \{ \{ ("Phil", "English"),$ $("Linda", "English"),$ $("Ralph", "English") \}, 13 \rangle$ |

This command changes R3's signature and state but leaves the relation's class unchanged. It illustrates two possible changes to a relation's signature, deletion of one attribute and addition of another attribute. Deletion of an attribute is usually expressed as a projection over the remaining attributes. Addition of an attribute requires that a value for the new attribute be determined for each tuple in the relation. Often, as in this example, a single default value is specified, which is then appended to each tuple. Note again that R3's state before transaction 13 has been discarded. □

The **modify_relation** command has several noteworthy properties. First, the command supports all update operations on a relation's state. Append is accommodated by an expression E , generally containing a union operator, that evaluates to a snapshot or historical state containing all the tuples in a relation's current state plus one or more tuples not in the relation's current state. Delete is accommodated by an expression E , generally containing a difference operator, that evaluates to a snapshot or historical state containing only a proper subset of the tuples in a relation's current state. Replace is accommodated by an expression E that evaluates to a snapshot or historical state that differs from a relation's current state only in the attribute values of one or more tuples.

Second, the **modify_relation** command ensures that a relation's class, signature, and state are consistent following update. The command changes a relation's state only if the new state is consistent with the relation's class and signature. Whenever the command changes a relation's signature, it also changes the relation's state to ensure consistency among the relation's class, signature, and state [Navathe & Fry 1976]. Likewise, whenever the command changes a relation's class, it also updates the relation's state, if necessary, to ensure consistency among the relation's class, signature, and state.

Finally, the **modify_relation** command always treats a relation's signature (state) sequence as an *append-only* sequence when the relation's current class is either rollback or temporal, but it does not automatically discard a relation's current signature (state) on update even if the relation's current class is snapshot or historical. If a relation's current class is a single-state class, the command discards the relation's current signature (state) on update only if the signature (state) is not part of the relation's history as a rollback or temporal relation.

Deleting a Relation

The command **destroy** assigns to a relation, whose current class is other than **UNDEFINED**, the new class **UNDEFINED**. It also deletes, either logically or physically, the relation's current signature and state.

```
C[destroy(I)](d, tn) =
  if      M = MSoT(d(I), tn) ∧ LastClass(d(I)) ≠ UNDEFINED
  then   (d[(M ||_3 ((UNDEFINED, tn, -)), ( ), ( ))/I], ok)
  else   (d, ERROR)
```

If the identifier I denotes a relation whose current class is other than **UNDEFINED**, the command simply appends the new class **UNDEFINED** to the relation's MSoT for the transaction in which the command occurs.

EXAMPLES.

C[destroy(R2)](DS, 14)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|--|---|--|
| R2 → | $\langle (\text{ROLLBACK}, 1, 5),$ $\quad\quad\quad (\text{ROLLBACK}, 9, 13),$ $\quad\quad\quad (\text{UNDEFINED}, 14, -) \rangle$ | $\langle ((\text{ename} \rightarrow \text{string},$ $\quad\quad\quad \text{ssn} \rightarrow \text{integer}), 1)$ $\quad\quad\quad () \rangle$ | $\langle (\emptyset, 1),$ $\quad\quad\quad \{(\text{Phil"}, 250861414),$ $\quad\quad\quad (\text{Linda"}, 147894290),$ $\quad\quad\quad (\text{Ralph"}, 459326889)\}, 3\rangle,$ $\langle (\emptyset, 9),$ $\quad\quad\quad \{(\text{Phil"}, 250861414),$ $\quad\quad\quad (\text{Linda"}, 147894290)\}, 11\rangle$ \rangle |

C[destroy(R3)](DS, 15)

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|---|-------------------|-------------------|
| R3 → | $\langle (\text{UNDEFINED}, 15, -) \rangle$ | $\langle \rangle$ | $\langle \rangle$ |

Because R2 denotes a relation whose current class is **ROLLBACK**, the first command uses the function *MSoT* to "close" the interval associated with the relation's current class. It then appends the element **(UNDEFINED, 14, -)** to R2's class sequence. These actions together

have the effect of logically deleting R_2 's current signature and state when transaction 14 commits. Note, however, that this signature and state information is still accessible via the rollback operator ρ . The second command uses the function $MSoT$ to physically delete R_3 's current class, signature, and state. No record of R_3 as a snapshot relation is retained. \square

It is important to observe from these, and previous, examples that signature and state information associated with a relation when its class was either snapshot or historical was transient. It was physically removed when it became outdated. Hence, the language is consistent with conventional relational DBMS's that discard out-of-date signature and state information (relation R_3 illustrates this). However, signature and state information associated with a relation when its class was rollback or temporal is retained, ensuring later access to past states via the rollback operator. Definition of the rollback operator assumes access to a complete record of a relation's signature and state during intervals when the relation's class was either rollback or temporal.

Renaming a Relation

The command `rename_relation` binds a relation's current class, signature, and state to a new identifier.

```
C[rename_relation(I1, I2)](d, tn) =
  if  (LastClass(d(I1)) ≠ UNDEFINED ∧ LastClass(d(I2)) = UNDEFINED
       ∧ Y[Y] = LastClass(d(I1)) ∧ Z[Z] = LastSignature(d(I1))
       ∧ C[define_relation(I2, Y, Z)](d, tn) = (d', ok)
       ∧ C[modify_relation(I2, *, *, I1)](d', tn) = (d'', ok)
       ∧ C[destroy(I1)](d'', tn) = (d''', ok))
    then (d''', ok)
  else (d, ERROR)
```

The `rename_relation` first assigns to the relation denoted by I_2 the current class and signature of the relation denoted by I_1 . It then assigns to I_2 the current state of I_1 . Finally, it assigns the class `UNDEFINED` to I_1 and deletes, either logically or physically, I_1 's current signature and state. Note that the execution environments for `rename_relation`'s three subordinate commands, while containing different database states, contain the same transaction number. Hence, the changes to both I_1 and I_2 become effective when a single transaction commits.

EXAMPLE. Recall that R_1 is the relation shown on page 62.

```
C[rename_relation(R1, R3)](DS, 16)
```

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|------|--|---|--------------|
| R1 → | $\langle (\text{ROLLBACK}, 2, 6),$ $\langle ((\text{surname} \rightarrow \text{string},$ $\text{ssn} \rightarrow \text{integer}), 2),$ $\langle (\text{surname} \rightarrow \text{string},$ $\text{class} \rightarrow \text{string}), 5)$ \rangle $(\text{UNDEFINED}, 16, -)\rangle$ | $\langle (\emptyset, 2),$ $\langle (\{\{\text{"Phil"}, 250861414\},$ $\{\text{"Linda"}, 147894290\},$ $\{\text{"Ralph"}, 459326889\}\}, 4),$ $\langle (\{\{\text{"Phil"}, \text{"junior"}\},$ $\{\text{"Linda"}, \text{"senior"}\},$ $\{\text{"Ralph"}, \text{"senior"}\}\}, 5)$ \rangle | |
| R3 → | $\langle (\text{SNAPSHOT}, 16, -)$ $\langle ((\text{ssn} \rightarrow \text{integer},$ $\text{class} \rightarrow \text{string}), 16)$ \rangle | $\langle (\{\{(250861414, \text{"junior"}),$ $(147894290, \text{"senior"}),$ $(459326889, \text{"senior"})\}, 16)\rangle$ | |

This command binds the current class, signature, and state of the relation denoted by R1 to the identifier R3. Hence, R3 becomes a snapshot relation when transaction 16 commits. The command also transforms R1 into an undefined relation, effective when transaction 16 commits. Because R1's current class, signature, and state are not part of the relation's history as either a rollback or temporal relation, they are physically deleted. □

A Sequence of Commands

If two or more commands appear in sequence, the commands are executed sequentially. If a command executes without error, the next command is executed using the database state resulting from the previous command's execution. If all the commands execute without error, the commands are mapped onto the final database state and the status code ok. If, however, any command's execution causes an error, the remaining commands are not executed and the status code ERROR is returned.

$$\mathbf{C}[C_1, C_2](d, tn) = \text{if } \mathbf{C}[C_1](d, tn) = (d', \text{ok}) \text{ then } \mathbf{C}[C_2](d', tn) \text{ else } (d, \text{ERROR})$$

Two or more commands appearing in sequence are all commands in the same transaction. Their execution environments have different database states but the same transaction number. Hence, if the commands change the same relation only the last changes to the relation's class, signature, and state are recorded in the final database state. Recall that while a relation's new class, signature, and state may depend on its current class, signature, and state, all commands define the resulting relation in terms of the relation's modified start of transaction. Also, if the commands change several relations, all the changes become effective when the transaction commits.

EXAMPLES. In the previous examples, we assumed that the commands were all taken from single-command transactions. We now show the result of executing multiple commands from the same transaction. Recall from page 86 that R2 is currently undefined.

```
C[define_relation(R2, rollback, (ename:string, ssn:integer)),
  modify_relation(R2, *, *, ρ(R2,5)),
  modify_relation(R2, *, *, R2 - σ ename="Linda" (R2))](DS, 17)
```

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|-----|--|---|--|
| R2→ | ((ROLLBACK, 1, 5), (ROLLBACK, 9, 13), (ROLLBACK, 17, -)) | ((ename → string, ssn → integer), 1) () | (((), 1), ({{("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3), ((), 9), ({{("Phil", 250861414), ("Linda", 147894290)}, 11), ({{("Phil", 250861414), ("Ralph", 459326889)}, 17})) |

```
C[destroy(R2), destroy(R3)](DS, 18)
```

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|-----|---|---|---|
| R2→ | ((ROLLBACK, 1, 5), (ROLLBACK, 9, 13), (ROLLBACK, 17, 17) (UNDEFINED, 18, -)) | ((ename → string, ssn → integer), 1) () | (((), 1), ({{("Phil", 250861414), ("Linda", 147894290), ("Ralph", 459326889)}, 3), ((), 9), ({{("Phil", 250861414), ("Linda", 147894290)}, 11), ({{("Phil", 250861414), ("Ralph", 459326889)}, 17)) |

| | <i>class</i> | <i>signature</i> | <i>state</i> |
|-----|----------------------|------------------|--------------|
| R3→ | ((UNDEFINED, 18, -)) | () | () |

In the first example, all three commands change R2. Yet, only the last changes to the relation's class, signature, and state are recorded in the database state. Although the first command defined R2 as a rollback relation and the other commands changed R2's state, only the final change in state is recorded. Hence, all the commands in a single transaction that change the same relation are treated as an atomic update operation. Note that temporary relations can be defined, modified, and then deleted within a transaction without their creation being recorded. In the second example, both R2 and R3 are deleted when transaction 18 commits. \square

4.2.6 Programs

The semantic function P defines the denotation of programs in our language, where a program is a sequence of one or more transactions. Transactions, in turn, may be either single-command or multiple-command transactions. P defines a program as a function that maps a database onto a database and a status code. A program is the only language construct that changes a database. Execution of a transaction that commits produces a new database and the status code OK, while execution of a transaction that aborts produces the original database unchanged and the status code ERROR.

$$P : PROGRAM \rightarrow [DATABASE \rightarrow [DATABASE \times \{OK, ERROR\}]]$$

Note that the environments for command and program execution, although similar, are different. The environment for command execution is a database state and the transaction number of the active transaction. In contrast, the environment for program execution is a database, which is an ordered pair consisting of a database state and the transaction number of the most recently committed transaction on that database state.

We now define formally the semantic function P for each kind of program allowed in our language.

$$\begin{aligned} P[\text{begin_transaction } C \text{ commit_transaction}](d, tn) = \\ \text{if } C[C](d, tn + 1) = (d', \text{OK}) \text{ then } ((d', tn + 1), \text{OK}) \text{ else } ((d, tn), \text{ERROR}) \end{aligned}$$

Committed transactions represent transactions that commit if their commands all execute without error. If all the commands in a transaction execute without error, the transaction is committed. The database's database-state component is updated to record the changes that the commands make to relations, the database's transaction-number component is incremented to record the transaction number of this most recently committed transaction, and the status code OK is produced. If any command's execution produces an error, the transaction is aborted. The database is left unchanged and the status code ERROR is produced. The database is valid independent of the status code.

$$P[\text{begin_transaction } C \text{ abort_transaction}](d, tn) = ((d, tn), \text{OK})$$

Aborted transactions are transactions, which the user initiates, that for some reason, dictated either by the user or by the system, abort rather than commit. They do not change the database.

$$\begin{aligned} P[P_1; P_2](d, tn) = \\ \text{if } P[P_1](d, tn) = ((d', tn'), \text{ok}) \text{ then } P[P_2](d', tn') \text{ else } P[P_2](d, tn) \end{aligned}$$

If a program contains multiple transactions, they are processed in sequence. If the first transaction commits and produces a new database, the second transaction is processed using the new database. Otherwise, the second transaction is processed using the original database.

Finally, we require that each arbitrary sequence of transactions representing a program map onto the database resulting from the execution of the transactions, in order, starting with the empty database. The empty database, (**EMPTY**, 0), is defined using the semantic function **EMPTY** : *IDENTIFIER* — (((**UNDEFINED**, 0, -)), (), ()). Hence, the database-state component of the empty database is defined to be the function that maps all identifiers onto undefined relations; the transaction-number component of the empty database is defined to be 0. This requirement is both necessary and sufficient to ensure that the transaction-number components of elements in the class, signature, and state sequences of each relation in the database are strictly increasing. A database will always be the cumulative result of all the transactions that have been performed on it since it was created.

We now define the semantic function **P'** that maps a program onto the database resulting from the execution of the program's transactions, starting with the empty database.

$$P' : \text{PROGRAM} \rightarrow \text{DATABASE}$$

$$P'[P] = \text{First}(P[P](\text{EMPTY}, 0))$$

where *First* is the function that maps an ordered pair onto the first component of the ordered pair.

4.2.7 Language Properties

We now state, as theorems, four properties of our algebraic language for database query and update, with informal proofs. The first property was stated initially as an objective of our extensions in Section 4.1.

Theorem 4.1 *The language is a natural extension of the relational algebra for database query and update.*

By natural extension, we mean that our semantics subsumes the expressive power of the relational algebra for database query and update. Expressions in our language are a strict superset of those in the relational algebra. Also, if we restrict the class of all relations to **UNDEFINED** and **SNAPSHOT**, then a natural extension implies that (a) the signature and state sequences of a defined relation will have exactly one element each: the relation's current signature and state; (b) a new state always will be a function of the current signature and state of defined relations via the relational algebra semantics; and (c) deletion will correspond to physical deletion.

PROOF. First, we show that expressions in our language are a strict superset of those in the relational algebra. Suppose we only allow expressions involving constants that denote snapshot states, identifiers that denote relations whose current class is **SNAPSHOT**, and the five relational operators. Then, expressions in the language are exactly those allowed in the relational algebra. But expressions in our language also may involve constants that denote historical states, identifiers that denote relations whose current class is other than **SNAPSHOT**, and both historical and rollback operators. Hence, expressions in our language are a strict superset of those in the relational algebra.

Next, we show that our semantics reduces to the conventional semantics of database state and database update via the relational algebra. Suppose we restrict the class of all relations to **UNDEFINED** and **SNAPSHOT**. Then,

- (a) The signature and state sequences of a defined relation will have exactly one element each, the relation's current signature and state. The relation can have no history as a rollback or temporal relation; hence its MSoT always will be $((), (), ())$. Because the **define_relation** and **modify_relation** commands change a relation's signature sequence by appending no more than one element to the relation's MSoT signature sequence, these commands always will produce a relation with a single-element signature sequence. The same holds for the relation's state sequence.
- (b) A new state always will be a function of the current signature and state of defined relations via the relational algebra semantics. Both the **define_relation** and **modify_relation** commands determine a new state via expression evaluation. The only semantically correct expressions are those involving constants that denote snapshot states, identifiers that denote relations whose current class is **SNAPSHOT**, and the five relational operators. These expressions are exactly those allowed in the relational algebra, their value depending on the current state and signature of defined relations only.
- (c) Deletion will correspond to physical deletion. The **destroy** command changes a relation by appending an element to the relation's MSoT class sequence; it never adds information to the relation's signature or state sequences. The **destroy** command always will produce a relation whose signature and state sequences are empty, which corresponds to physical deletion of a relation's current signature and state. ■

Theorem 4.2 *The language is a natural extension of our historical algebra for database query and update.*

PROOF. An argument, analogous to that given above for the snapshot algebra, holds. ■

The third property argues that the semantics is minimal, in a specific sense. Other definitions of minimality, such as minimal redundancy or minimal space requirements, are more appropriate for the physical level, where actual data structures are implemented, than for the algebraic level.

Theorem 4.3 *The semantics of the language minimizes the number of elements in a relation's class, signature, and state sequence needed to record the relation's current class, signature, and state and its history as a rollback or temporal relation.*

PROOF. Assume that the number of elements in a relation's class sequence exceeds the minimum needed to record the relation's current class and its history as a rollback or temporal relation. Then, (a) there are two consecutive elements in the sequence that can be combined or (b) there is an element in the sequence that can be removed. Consider case (a). Consecutive elements in the class sequence can be combined only if they record the same class over non-disjoint intervals. But the commands only append a new element to a relation's class sequence if it either differs from the relation's MSoT class or its interval is disjoint from that of the relation's MSoT class. Hence, no two consecutive elements in a relation's class sequence can have the same class but non-disjoint intervals. Now, consider case (b). Commands always produce a new relation by appending new class information to a relation's MSoT class sequence. But, it can be shown that all elements in a relation's MSoT class sequence record intervals when the relation was either a rollback or temporal relation. Hence, no element can be removed. If no two elements can be combined and no element can be removed, our assumption is contradicted and the number of elements in the class sequence must be minimal. Similar arguments hold for the relation's signature and state sequences. ■

The fourth property ensures that the language accommodates implementations that use WORM optical disk to store non-current class, signature, and state information, another objective of our extensions.

Theorem 4.4 *Transactions change only a relation's class, signature, and state current at the start of the transaction.*

PROOF. This property is a consequence of the way the *MSoT* function is defined and used. We first prove the property for a relation's signature sequence and then for its class and state sequences.

A relation's current signature at the start of a transaction is the last element in the relation's signature sequence. Assume, therefore, that a transaction changes an element that is in the relation's signature sequence at the start of the transaction but is not the last element in the sequence. Such a change must occur during the execution of a command. When the first command in a transaction executes, *MSoT* discards the last element in

the relation's signature sequence, if the relation's current class is either SNAPSHOT or HISTORICAL. Otherwise, it retains all the elements. When each subsequent command in the transaction is executed, *MSoT* only discards any element that the preceding command added to the sequence. Hence, *MSoT* never changes an element in a relation's signature sequence that precedes the last element in the sequence at the start of the transaction. Commands, although they may append an element to the relation's *MSoT* signature sequence, never change existing elements. Hence, commands never change an element in a relation's signature sequence that precedes the last element in the sequence at the start of the transaction and our assumption is contradicted. The same argument holds for the relation's state sequence.

The above argument holds for a relation's class sequence with the following provisos. When the first command in a transaction executes, *MSoT* discards the last element in the relation's class sequence if the relation's current class is UNDEFINED. Also, if the relation's current class is either ROLLBACK or TEMPORAL, *MSoT* changes the last element in the sequence to "close" the interval assigned to the relation's current class at the start of the transaction. When each subsequent command in the transaction is executed, *MSoT* "re-closes" this same interval, if extended by the preceding command. Hence, *MSoT* never changes an element in a relation's class sequence that precedes the last element in the sequence at the start of the transaction. Commands may change the last element in a relation's *MSoT* class sequence to "extend" the interval assigned to the class component of that element, but only if the new class and the relation's *MSoT* class are equal and their intervals abut. This occurs only when the last element in the relation's *MSoT* class sequence corresponds to the last element in the relation's class sequence at the start of the transaction (i.e., the class of the relation at the start of the transaction was either ROLLBACK or TEMPORAL). Otherwise, the intervals could not abut as there would exist an intervening interval when the relation's class was either SNAPSHOT, HISTORICAL, or UNDEFINED. Hence, commands never change an element in a relation's class sequence that precedes the last element in the sequence at the start of the transaction. ■

4.3 Additional Aspects of the Rollback Operators

The rollback operators in our language are more powerful than suggested in the previous section, in several ways. First, the rollback operators, as defined, are restricted to the retrieval of a single snapshot or historical state from a named relation current at the time of a specified transaction. In reality, however, the rollback operators derive a single snapshot or historical state from one or more of the named relation's stored states rather than simply retrieving a single state. The rollback operators actually roll back a relation to the subsequence of the relation's state sequence corresponding to an interval of time of arbitrary length, if the relation's class and signature remained constant over that interval of time. The rollback operators return the single state composed of tuples from all the states in the specified subsequence of relation states (effectively, a relational union, either snapshot or historical, is performed). The rollback operators thus take two transaction

times as arguments:

$$E ::= \rho(I, N, N) \mid \hat{\rho}(I, N, N)$$

Second, the rollback operators do not simply retrieve a snapshot or historical state from a named relation but rather an augmented version of that state. To the state's explicit attributes, defined in its signature, the rollback operators add new explicit attributes corresponding to the state's implicit time attributes (i.e., transaction times for snapshot states, transaction and valid times for historical states). The rollback operators' addition of these new attributes to the state's existing explicit attributes allows the user to display the values of the state's implicit time attributes without allowing direct access to the attributes themselves. These explicit values are considered to be in the domain of user-defined time. This behavior requires that the semantic function T compute a relational signature containing these additional attributes.

Third, the rollback operator ρ can be applied to temporal relations as well as rollback relations. If ρ rolls back a relation to a time when the relation's class was TEMPORAL, ρ will convert the relation's historical state current at that time into a corresponding snapshot state and return this new snapshot state. Likewise, the rollback operator $\hat{\rho}$ can be applied to rollback relations as well as temporal relations. If $\hat{\rho}$ rolls back a relation to a time when the relation's class was ROLLBACK, $\hat{\rho}$ will convert the relation's snapshot state current at that time into a corresponding historical state and return this new historical state.

While these extensions are conceptually straightforward, the notation required to define them formally is cumbersome and will not be presented.

4.4 Summary and Related Work

In summary, we have defined in this chapter an algebraic language for database query and update. It subsumes both the relational algebra and our historical algebra, and it supports both snapshot and historical rollback. The language also has a simple semantics and supports scheme evolution. Only two additional operators, ρ and $\hat{\rho}$, were necessary. The additions required for transaction time did not compromise any of the useful properties of the conventional snapshot algebra or our historical algebra. Type-checking was introduced, freeing the encapsulated algebras from dealing with expressions not consistent with the (possibly time-varying) scheme. Also, the approach introduced here is not restricted to the relation algebra and our historical algebra. It can accommodate most historical algebras; we only require that expressions in the algebra evaluate to historical states.

This chapter makes three contributions. The primary contribution is an algebraic means of supporting both scheme and contents evolution in the context of general support for transaction time. As an algebraic language for database query and update, our language can serve as the underlying evaluation mechanism for queries and updates in a temporal data manipulation language that supports evolution of a database's contents and scheme.

It can also be used as the basis for proving various physical implementations of temporal database management systems correct. Our language also is compatible with efforts to add transaction time to the relational data model at both the user-interface and the physical levels. At least three temporal query languages have been proposed that support rollback operations [Ariav 1986, Ben-Zvi 1982, Snodgrass 1987] and several studies have investigated efficient storage and access strategies for temporal databases [Ahn 1986A, Ahn 1986B, Ahn & Snodgrass 1986, Ahn & Snodgrass 1988, Lum et al. 1984, Rotem & Segev 1987, Shoshani & Kawagoe 1986, Thirumalai & Krishna 1988]. Also, the considerable research into efficient storage and access strategies for persistent data structures [Chazelle 1985, Cole 1986, Dobkin & Munro 1985, Myers 1984, Sarnak & Tarjan 1986] can be used to implement our semantics. Verma and Lu discuss the use of persistent data structures to implement databases containing either rollback or temporal relations [Verma & Lu 1987].

The second contribution is the model of database state as a sequence ordered by transaction time. Each element in the sequence is a cross-section of the database state at a transaction, containing, for each relation defined at that time, either a snapshot or historical state. In a related effort, Abiteboul and Vianu have defined a transaction language TL consisting of parameterized expressions containing tuple insertions and deletions and a looping construct [Abiteboul & Vianu 1987]. In TL, the database state is modeled "procedurally" by providing the transaction(s) that compute that state; transaction time is implicit. The focus of this and previous research [Abiteboul & Vianu 1985, Abiteboul & Vianu 1986, Vianu 1983] is developing a characterization of the possible database states computable by constrained transactions, with the goal of using such transactions as a specification tool for stating dynamic constraints. The goal of our language is different; we model the evolution of the database in terms of transactions specified by the user in a calculus-based update language that is translated by the DBMS into algebraic expressions.

The third contribution is the formalization of the evolving state through the definition of the `modify_relation` command. This aspect has been investigated at the user-interface level by several researchers in the context of dynamic constraints on updates of database instances [Brodie 1981, Ceri et al. 1981, Hammer & McLeod 1981]. At the algebraic level, only Ben-Zvi has attempted such a formalization. His approach is to provide procedures for various manipulation commands (e.g., insert, delete, terminate) and prove that these procedures maintain various desirable properties. The effect of these procedures are localized to a specific tuple that changes during the transaction. Our `modify_relation` command simply replaces or appends a new entire snapshot or historical state, allowing many tuples to change during a transaction. Of course, actual implementations would be based on more complex representations that exhibit greater space and time efficiency. Verifying the correctness of such implementations would involve demonstrating the equivalence of their semantics with the simple semantics presented here.

There have been two other attempts to incorporate both valid time and transaction time in an algebra. In Ben-Zvi's proposal, valid time and transaction time were supported through the addition of implicit time attributes to each tuple in a relation [Ben-Zvi 1982]. The algebra was extended with the *Time-View* algebraic operator which takes a relation and two times as arguments and produces the subset of tuples in the relation valid at

the first time (the valid time) as of the second time (the transaction time). The Time-View operator thus rolls back a relation to a transaction time but returns only a subset of the tuples in the relation at that transaction time (i.e., those tuples valid at some specified time). This restricted definition of the Time-View operator is tied inextricably to his particular handling of valid time. Our approach is compatible with any historical algebra. Gadia represents valid time and transaction time as two symmetrical dimensions in a boolean algebra of multidimensional time stamps [Gadia & Yeung 1988]. He allows rollback operations on transaction time through a generalized restriction operator, which may be applied to any of a relation's time dimensions. He does not, however, address the problems of database update or scheme evolution.

While a few authors have envisaged the benefits of a time-varying scheme [Ariav 1986, Ben-Zvi 1982, Shiftan 1986, Woelk et al. 1986], only one other extension of the relational algebra, that proposed by Ben-Zvi, includes support for an evolving scheme. Ben-Zvi proposes that a temporal relation's scheme itself be represented as a temporal relation, thus providing a uniform treatment for evolution of a relation and its scheme [Ben-Zvi 1982]. He does not, however, provide formal semantics for scheme evolution in the context of general support for transaction time. Martin proposes a non-algebraic solution to the problem of an evolving scheme in temporal databases using modal temporal logic [Martin et al. 1987]. A scheme temporal logic is proposed to deal with changes in scheme. A set of scheme temporal logic formulae are associated with a scheme to describe its evolution and temporal queries are interpreted in the context of these formulae. This approach, unlike ours, forces synchronization between valid time and scheme changes. Again, formal semantics are not provided. Finally, Adiba, in describing mechanisms for the storage and manipulation of historical multi-media data, advocates, like Ben-Zvi, that the history notion used to model changes in a database's contents also be used to model changes in the database's scheme [Adiba & Bui Quang 1986].

While there has been significant interest in database reorganization and restructuring [Banerjee et al. 1987, Markowitz & Makowsky 1987, Navathe & Fry 1976, Navathe 1980, Roussopoulos & Mark 1985, Shu et al. 1977, Shu 1987, Sockut & Goldberg 1979], such approaches have assumed that the scheme (and hence the contents) of the entire database will be modified during restructuring, ensuring that only one scheme is in force. Since we formalize the scheme as a sequence ordered by transaction time, several schemes can be in force, selectable through the rollback operator. A second difference is that we focus solely on algebraic support for scheme evolution, while the other papers considered the related issues of determining what changes to the scheme are necessary and what those changes imply regarding the new state to be calculated. Certainly, all these issues must be addressed before a comprehensive solution to scheme evolution is developed.

In contrast to these previous approaches, the WAND system did permit several generations of schemes to be simultaneously present [Gerritsen & Morgan 1976]. This system differs from our approach in two respects. First, the WAND system was based on the network model, whereas our approach is based on the relational model. More significantly, scheme evolution was supported in the WAND system to allow dynamic restructuring of the database. While data in the WAND system could also be associated with one of sev-

eral generations of schemes, the data were always restructured to match the most recent scheme as they were referenced. Multiple generations were introduced to achieve concurrency between restructuring and execution of application programs. Hence, the underlying model did not support transaction time or rollback. The WAND system was effectively a snapshot DBMS that permitted applications to access and change the database while a global restructuring was being performed.

ORION, a prototype object-oriented database system being developed at MCC, takes a similar approach [Banerjee et al. 1987]. An important difference is that when the scheme in ORION is modified, no disk-resident data instances need be updated. Instead, when an instance is referenced by an application program and fetched into memory, it is transformed into an instance conforming to the scheme currently in effect. Again, only one scheme is ever in effect; the implementation places the burden of updating the data across a scheme change on subsequent retrievals.

Several researchers have used denotational semantics to define formally the semantics of databases, DBMS's, and query languages. Subieta proposes an approach for defining query languages formally using denotational semantics [Subieta 1987]. This approach allows powerful query languages with precise semantics to be defined for most database models. Rishe proposes that denotational semantics be used to provide a uniform treatment of database semantics at different information levels based on hierarchies of domains of mappings from "less semantic" representations of information into "more semantic" representations [Rishe 1985]. Neither Subieta nor Rishe, however, include in their approaches any facilities for dealing with transaction time or an evolving scheme. Lee proposes a denotational semantics for administrative databases, where databases are regarded as a collection of logical assertions [Lee 1985]. Here, the denotation of an expression in a first-order predicate calculus is based, in part, on its evaluation in a time dimension, analogous to valid time, in a possible world, analogous to a cross-section of a database state at a transaction.

Chapter 5

Equivalence With TQuel

In Chapter 3 we extended the snapshot algebra to handle valid time by defining a historical algebra. Then, in Chapter 4 we described an approach for adding transaction time to both the snapshot algebra and our historical algebra. We now show that the algebraic language for query and update of temporal databases defined in those chapters has the expressive power of *TQuel* (*Temporal QUEry Language*) [Snodgrass 1987]. *TQuel* is a version of *Quel* [Held et al. 1975], the calculus-based query language for the *Ingres* relational database management system [Stonebraker et al. 1976], augmented to handle both valid time and transaction time. A brief review of *TQuel* constructs for handling time appears in Section 2.1.

Because our formalization of the contents of a database differs from that used in *TQuel*'s semantics, we first show the correspondence between a *TQuel* database and a database as defined in our language. We then show that our language subsumes *TQuel* by giving, for each type of *TQuel* statement (i.e., *retrieve*, *create*, *append*, *replace*, *delete*, and *destroy*), its equivalent transaction in the language. (We postpone discussion of views until the next chapter). We first consider the basic *TQuel* *retrieve* statement without aggregates and then more complex *TQuel* *retrieve* statements with aggregates in their target lists, where clauses, and when clauses. Having dispensed with the *retrieve* statement, we proceed to give the language equivalences for the *TQuel* *create*, *append*, *replace*, *delete*, and *destroy* statements. We conclude the chapter with two language correspondence theorems.

For notational convenience, we associate " " with *TQuel* database states, relation states, tuple variables, and expressions throughout this chapter to differentiate them from their counterparts in our language. We also consider only databases of temporal relations, the most general class of relations. All arguments apply equally to databases containing snapshot, historical, and rollback relations.

5.1 TQuel Database

As in our language, a *TQuel* database can be viewed as an ordered pair consisting of a database state and the transaction number of the most recently committed transaction

on the database. Similarly, a TQuel database state can be viewed as a mapping from identifiers onto relations. Relations, however, are defined differently in the two languages. Unlike our language, which represents a relation's contents as a sequence of relation states indexed by transaction time, the formal semantics of TQuel conceptually embeds a relation's contents, whether the relation's class be snapshot, rollback, historical, or temporal, in a single snapshot state. The embedding is done purely for convenience in developing the semantics. TQuel, unlike our language, assumes tuple time-stamping. It represents valid time by adding two implicit attributes to each tuple to specify the time when the tuple became valid (i.e., *From*) and the time when the tuple became invalid (i.e., *To*). TQuel represents transaction time by adding two more implicit attributes to each tuple to specify the time when the tuple was entered into the relation (i.e., *Start*) and the time when the tuple was removed from the relation (i.e., *Stop*).

EXAMPLE. Assume that we are given the historical state S_1 from page 25 over the relation signature *Student* with the attributes {*sname*, *course*}, duplicated below.

$$\{ \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}) \rangle, \\ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \\ \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}) \rangle \}$$

This historical state, if represented as a TQuel embedded temporal relation created by transaction 423, would have the following form.

| <i>sname</i> | <i>course</i> | <i>From</i> | <i>To</i> | <i>Start</i> | <i>Stop</i> |
|--------------|---------------|-------------|-----------|--------------|-------------|
| "Phil" | "English" | 1 | 2 | 423 | ∞ |
| "Phil" | "English" | 3 | 5 | 423 | ∞ |
| "Norman" | "English" | 1 | 3 | 423 | ∞ |
| "Norman" | "Math" | 5 | 7 | 423 | ∞ |

We show the TQuel relation as a table simply for notational convenience in identifying the implicit attributes. Note that in TQuel a tuple's interval of validity doesn't include the chronon assigned to the attribute *To*. \square

As shown in this example, TQuel, unlike our language, allows value-equivalent tuples (i.e., tuples with identical values for their explicit attributes) in a relation state. It assumes, however, that value-equivalent tuples, active at the time of a transaction *tn*, are *coalesced*; they neither overlap nor are adjacent in time. We define here the boolean function *Coalesced* that determines whether a TQuel embedded temporal relation is coalesced. For this definition, let R' be a TQuel embedded temporal relation with explicit attributes $A = \{I_1, \dots, I_m\}$.

Coalesced(R') =

$$\begin{aligned} & \forall tn \forall r'_1 \forall r'_2, (r'_1 \in R' \wedge r'_2 \in R' \\ & \quad \wedge \forall I, I \in \mathcal{A}, r'_1(I) = r'_2(I) \\ & \quad \wedge r'_1(\text{Start}) \leq tn < r'_1(\text{Stop}) \wedge r'_2(\text{Start}) \leq tn < r'_2(\text{Stop})) \\ & \quad (r'_1(\text{To}) < r'_2(\text{From}) \vee r'_2(\text{To}) < r'_1(\text{From})) \end{aligned}$$

We now show that it is possible to map the embedded, coalesced temporal relations used in TQuel's formal semantics onto historical relation states in our language. The transformation function TF_T maps a TQuel embedded temporal relation R' with attributes $\mathcal{A}_{R'} = \{I_1, \dots, I_m, \text{From}, \text{To}, \text{Start}, \text{Stop}\}$ and a transaction number tn onto R' 's equivalent historical state R at the time of transaction tn , where R is a historical state with attributes $\mathcal{A}_R = \{I_1, \dots, I_m\}$ in our language.

$$\begin{aligned} TF_T(R', tn) \triangleq & \{ u^m \mid (\forall I, I \in \mathcal{A}_R, \forall t, t \in \text{Valid}(u(I)), \\ & \exists r', (r' \in R' \\ & \quad \wedge \forall I', I' \in \mathcal{A}_R, \text{Value}(u(I')) = r'(I') \\ & \quad \wedge \text{Before}(\text{Pred}(r'(\text{Start})), tn) \wedge \text{Before}(tn, r'(\text{Stop})) \\ & \quad \wedge t \in \text{Extend}(r'(\text{From}), \text{Pred}(r'(\text{To}))) \\ & \quad) \\ & \quad) \\ & \quad \wedge (\forall r', (r' \in R' \\ & \quad \wedge \forall I, I \in \mathcal{A}_R, r'(I) = \text{Value}(u(I)) \\ & \quad \wedge \text{Before}(\text{Pred}(r'(\text{Start})), tn) \wedge \text{Before}(tn, r'(\text{Stop}))), \\ & \quad \forall I, I \in \mathcal{A}_R, \text{Extend}(r'(\text{From}), \text{Pred}(r'(\text{To}))) \subseteq \text{Valid}(u(I)) \\ & \quad)) \} \end{aligned}$$

where *Before* is the " $<$ " predicate on integers. The first clause of this definition ensures that each tuple in $TF_T(R', tn)$ has at least one value-equivalent tuple in R' that was active at transaction tn (i.e., $\text{Before}(\text{Pred}(r'(\text{Start})), tn) \wedge \text{Before}(tn, r'(\text{Stop}))$). The second clause in the definition ensures that each subset of value-equivalent tuples in R' , active at transaction tn , is represented by a single tuple in $TF_T(R', tn)$. Note that the same time-stamp is assigned to each attribute of a tuple in $TF_T(R', tn)$. This time-stamp is simply

the union of the time-stamps of those value-equivalent tuples in R' active at transaction tn . We could define, without difficulty, analogous transformation functions TF_H and TF_R for TQuel embedded historical and rollback relations. A transformation function is not required for snapshot relations because a TQuel snapshot relation is formalized identically to our snapshot state.

Because TQuel assumes that value-equivalent tuples are coalesced, the valid times assigned value-equivalent tuples in R' , active at transaction tn , are disjoint, non-adjacent intervals. Hence, each distinguishable interval in the attribute time-stamps of a tuple in $TF_T(R', tn)$ corresponds to the valid time of one of the tuple's value-equivalent counterparts in R' , as we now show.

Lemma 5.1 $\forall r, r \in TF_T(R', tn), \forall I, I \in \mathcal{A}_R, \forall IN, IN \in Interval(Valid(r(I)))$.

$$\begin{aligned} & \exists r', (r' \in R' \\ & \wedge \forall I', I' \in \mathcal{A}_R, Value(r(I')) = r'(I') \\ & \wedge Before(Pred(r'(\text{Start})), tn) \wedge Before(tn, r'(\text{Stop})), \\ & \wedge IN = Extend(r'(\text{From}), Pred(r'(\text{To}))) \\ &) \end{aligned}$$

PROOF. Apply the definitions of *Coalesced* and *Interval* to TF_T and simplify. ■

EXAMPLE. If we let R' be the TQuel embedded temporal relation given in the previous example, then $TF_T(R', 423)$ is the historical state S_1 , also given in the example. Consider the following tuple r taken from the historical state and the tuples r'_1 and r'_2 taken from the TQuel embedded temporal relation.

$$\begin{aligned} r &= ((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\})) \\ r'_1 &= (\text{"Phil"}, \text{"English"}, 1, 2, 423, \infty) \\ r'_2 &= (\text{"Phil"}, \text{"English"}, 3, 5, 423, \infty) \end{aligned}$$

then

$$\begin{aligned} Interval(Valid(r(\text{pname}))) &= \{\{1\}, \{3, 4\}\} \\ Interval(Valid(r(\text{state}))) &= \{\{1\}, \{3, 4\}\} \\ Extend(r'_1(\text{From}), Pred(r'_1(\text{To}))) &= \{1\} \\ Extend(r'_2(\text{From}), Pred(r'_2(\text{To}))) &= \{3, 4\} \end{aligned}$$

□

TQuel does not allow changes to the signature of an embedded temporal relation. once the relation is created. Hence, we can define a TQuel database (d', tn) of embedded temporal relations to be *temporally equivalent* to the database (d, tn) in our language if. and only if,

```

 $\forall I, I \in IDENTIFIER, \forall tn', 1 \leq tn' \leq tn.$ 

if  $Findclass(d(I), tn') \neq \text{ERROR}$ 

then  $(Findclass(d(I), tn') = \text{TEMPORAL} \wedge Class(d'(I), tn') = \text{TEMPORAL}$ 
       $\wedge FindSignature(d(I), tn') = Signature(d'(I), tn')$ 
       $\wedge Findstate(d(I), tn') = TF_T(d'(I), tn'))$ 

else  $Class(d'(I), tn') = \text{ERROR}$ 

```

where the function *Class* returns the class of a TQuel embedded relation at the time of a specified transaction and *Signature* returns the signature that corresponds to the relation's explicit attributes. These functions can be defined analogously to the functions *Findclass* and *FindSignature* in our language.

We also define a TQuel statement and a transaction in our language to be equivalent if, and only if, they map temporally equivalent databases onto temporally equivalent databases.

5.2 TQuel Retrieve Statement

Assume that we are given the TQuel database (d', tn) containing the k embedded temporal relations R'_1, \dots, R'_k on signatures z'_1, \dots, z'_k that induce. respectively, the attributes.

$$\mathcal{A}_1 = \{I_{1,1}, \dots, I_{1,m_1}, \text{From}, \text{To}, \text{Start}, \text{Stop}\}$$

...

$$\mathcal{A}_k = \{I_{k,1}, \dots, I_{k,m_k}, \text{From}, \text{To}, \text{Start}, \text{Stop}\}$$

For notational convenience, assume that $I_{1,1}, \dots, I_{k,m_k}$ are unique. Furthermore, let i_1, i_2, \dots, i_n be integers, not necessarily distinct, in the range 1 to k and a_l , $1 \leq l \leq n$, be a distinct integer in the range 1 to m_{i_l} . Then, the TQuel retrieve statement has the following syntax

```

range of  $r'_1$  is  $I_1$ 
...
range of  $r'_k$  is  $I_k$ 
retrieve into persistent  $I_{k+1}$  ( $I_{k+1,1} = r'_{i_1}.I_{i_1,a_1}, \dots, I_{k+1,n} = r'_{i_n}.I_{i_n,a_n}$ )
    valid from  $v'$  to  $\chi'$                                 (5.1)
    where  $\psi'$ 
    when  $r'$ 
    as of  $\alpha'$ 
```

where $d'(I_j)$, $1 \leq j \leq k$, denotes the embedded temporal relation R'_j . We assume the type correctness of this statement for the TQuel database (d', tn) . The statement, when executed on (d', tn) , creates a new relation denoted by I_{k+1} , computes a new embedded temporal relation R'_{k+1} with attributes

$$A_{k+1} = \{I_{k+1,1}, \dots, I_{k+1,n}, \text{From}, \text{To}, \text{Start}, \text{Stop}\}$$

and changes the database state d' to map I_{k+1} onto this new relation. Execution of the statement also causes the transaction-number component of the database to be incremented.

5.2.1 Semantics

The tuple calculus for the new relation is

$$\begin{aligned}
R'_{k+1} \triangleq & \{ u^{n+4} \mid (\exists r'_1) \cdots (\exists r'_k) \\
& (r'_1 \in R'_1 \wedge \cdots \wedge r'_k \in R'_k) \\
& \wedge u(I_{k+1,1}) = r'_{i_1}(I_{i_1,a_1}) \wedge \cdots \wedge u(I_{k+1,n}) = r'_{i_n}(I_{i_n,a_n}) \\
& \wedge u(\text{From}) = \Phi'_v((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To}))) \\
& \wedge u(\text{To}) = \Phi'_x((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To}))) \\
& \wedge u(\text{Start}) = \text{current transaction number} \wedge u(\text{Stop}) = \infty \\
& \wedge \text{Before}(u(\text{From}), u(\text{To})) \\
& \wedge \Psi'_\psi(r'_1(I_{1,1}), \dots, r'_k(I_{k,m_k})) \\
& \wedge \Gamma'_\tau((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To}))) \\
& \wedge \forall j, 1 \leq j \leq k, \text{Before}(\text{Pred}(r'_j(\text{Start})), \Phi'_\alpha) \wedge \text{Before}(\Phi'_\alpha, r'_j(\text{Stop})) \\
& \}
\end{aligned} \tag{5.2}$$

where $\text{Coalesced}(R'_1), \dots, \text{Coalesced}(R'_{k+1})$ are true, the ordered pair $(r'_j(\text{From}), r'_j(\text{To}))$, $1 \leq j \leq k$, represents the interval $[r'_j(\text{From}), r'_j(\text{To})]$, and $\Psi'_\psi, \Phi'_v, \Phi'_x, \Phi'_\alpha$, and Γ'_τ , are the denotations described below of ψ' , v' , x' , τ' , and α' respectively. Φ'_α does not require any parameters because, unlike Φ'_v and Φ'_x , it can't contain tuple variables.

Ψ'_ψ is obtained by replacing each occurrence of an attribute reference $r'_j.I_{j,a}$, $1 \leq j \leq k$, $1 \leq a \leq m_j$, in ψ' with $r'_j(I_{j,a})$ and each occurrence of a logical operator with its corresponding logical predicate. That is,

$$r'_j.I_{j,a} \rightarrow r'_j(I_{j,a}),$$

and $\rightarrow \wedge$,

or $\rightarrow \vee$, and

not $\rightarrow \neg$.

Φ'_v , Φ'_x , and Φ'_α are obtained by replacing each occurrence of a tuple variable r'_j in v' and x' with the ordered pair $(r'_j(\text{From}), r'_j(\text{To}))$ and each occurrence of a temporal constructor with a corresponding function. That is,

$$r'_j \rightarrow (r'_j(\text{From}), r'_j(\text{To}))$$

begin of IN $\rightarrow \text{beginof}(IN)$,

end of IN $\rightarrow \text{endof}(IN)$,

IN₁ overlap IN₂ $\rightarrow \text{overlap}(IN_1, IN_2)$, and

IN₁ extend IN₂ $\rightarrow \text{extend}(IN_1, IN_2)$

where *beginof*, *endof*, *overlap*, and *extend* are functions on the domain \mathcal{IN} . Formal definitions for these functions are presented elsewhere [Snodgrass 1987].

Γ'_τ is obtained by replacing each occurrence of a logical operator in τ' with its corresponding logical predicate according to the rules given for its replacement in ψ' , replacing each occurrence of a tuple variable or temporal constructor according to the rules given for their replacement in v' and χ' , and replacing each occurrence of a temporal predicate operator with an analogous predicate on intervals. That is,

$IN_1 \text{ precede } IN_2 \rightarrow \text{precede}(IN_1, IN_2)$,

$IN_1 \text{ overlap } IN_2 \rightarrow \text{overlap}(IN_1, IN_2)$, and

$IN_1 \text{ equal } IN_2 \rightarrow \text{equal}(IN_1, IN_2)$

where *precede*, *overlap*, and *equal* are predicates on the domain \mathcal{IN} . Formal definitions for these predicates are presented elsewhere [Snodgrass 1987].

Before we present the algebraic equivalence of the TQuel retrieve statement, we describe the mapping of each of the TQuel syntactic constructs ψ' , v' , χ' , α' , and τ' onto its counterpart in our language. ψ is obtained by replacing each occurrence of $r'_j.I_{j,a}$, $1 \leq j \leq k$, $1 \leq a \leq m_j$, in ψ' with $I_{j,a}$. v , χ , and α are obtained by replacing each occurrence of a tuple variable r'_j , $1 \leq j \leq k$, in v' and χ' with $I_{j,1}$ and each occurrence of a temporal constructor with its algebraic equivalence. That is,

$r'_j \leftrightarrow I_{j,1}$,

begin of $IN \rightarrow \text{First}(IN)$,

end of $IN \rightarrow \text{Last}(IN)$,

$IN_1 \text{ overlap } IN_2 \rightarrow IN_1 \cap IN_2$, and

$IN_1 \text{ extend } IN_2 \rightarrow \text{Extend}(\text{First}(IN_1), \text{Last}(IN_2))$.

τ is obtained by replacing each occurrence of a tuple variable or temporal constructor in τ' according to the rules given for their replacement in v' and χ' , and replacing each occurrence of a temporal predicate operator with its algebraic equivalence. That is,

$IN_1 \text{ precede } IN_2 \rightarrow \text{Last}(IN_1) < \text{First}(IN_2) \text{ or } \text{Last}(IN_1) = \text{First}(IN_2)$,

$IN_1 \text{ overlap } IN_2 \rightarrow \text{not } (IN_1 \cap IN_2 = \{ \})$, and

$IN_1 \text{ equal } IN_2 \rightarrow IN_1 = IN_2$.

Note from the definition of $TF_T(R', tn)$ that a tuple in $TF_T(R', tn)$ has the same time-stamp for each of its attributes. Hence, although we require that each occurrences of a tuple variable (r'_j in v' , χ' , α' , and τ' be replaced with the same attribute name (i.e., $I_{j,1}$), we could have specified any attribute of historical state R_j .

The semantic functions that map ψ , v , χ , α , and τ onto their denotations in our language are defined in Appendix B. Let Ψ_ψ , Φ_v , Φ_χ , Φ_α , and Γ_τ be the denotations of ψ , v , χ , α , and τ , respectively. Then, the following two lemmas, which will be needed in the equivalence proof to be presented shortly, hold.

Lemma 5.2 Φ_v , Φ_χ , Φ_α , and Γ_τ are semantically equivalent to Φ'_v , Φ'_χ , Φ'_α , and Γ'_τ , respectively. That is, the result of evaluating Φ'_v , Φ'_χ , Φ'_α , and Γ'_τ for tuples r'_j , $r'_j \in R'_j$, $1 \leq j \leq k$, is the same as the result of evaluating Φ_v , Φ_χ , Φ_α , and Γ_τ for the intervals IN_j , $IN_j = Extend(r'_j(\text{From}), Pred(r'_j(\text{To})))$ substituted for the attribute name $I_{j,1}$.

PROOF. The semantic equivalence follows directly from the definitions of the functions that are used in Φ'_v , Φ'_χ , Φ'_α , and Γ'_τ [Snodgrass 1987] and the functions, defined in Appendix B, that are used in Φ_v , Φ_χ , Φ_α , and Γ_τ . ■

Lemma 5.3 $t \in Extend(\Phi'_v(\dots), Pred(\Phi'_\chi(\dots))) \rightarrow Before(\Phi'_v(\dots), \Phi'_\chi(\dots))$.

PROOF. It follows directly from the definition of *Extend*, given in Appendix B, that $t \in Extend(\Phi'_v(\dots), Pred(\Phi'_\chi(\dots)))$ implies $\Phi'_v(\dots) \leq t < \Phi'_\chi(\dots)$, which in turn implies $Before(\Phi'_v(\dots), \Phi'_\chi(\dots))$. ■

5.2.2 Correspondence Theorem

Having defined the algebraic equivalences of expressions in the new TQuel clauses, we can now define the algebraic equivalence of a TQuel retrieve statement. Assume that we are given the value domains D_u , $1 \leq u \leq e$, and the semantic function **DN**, defined in Appendix B, that maps identifiers onto value domains (i.e., **DN** “names” value domains). Let u_1, u_2, \dots, u_n be integers, not necessarily distinct, in the range 1 to e where signature $z_{i,l}$ maps attribute I_{i,l,a_l} , and **DN** maps domain name I_{u_l} , onto value domain D_{u_l} , $1 \leq l \leq n$. Then the algebraic equivalence of the TQuel retrieve statement, without aggregates, is

```

begin_transaction

define_relation(I_{k+1}, temporal, (I_{k+1,1}:I_{u_1}, ..., I_{k+1,n}:I_{u_n})),
modify_relation(I_{k+1}, *, *, \pi(I_{k+1,1} := (I_{i_1,a_1} \otimes I_{i_1,a_1}), ..., I_{k+1,n} := (I_{i_n,a_n} \otimes I_{i_n,a_n}))(

\delta\tau, (I_{1,1} := Extend(v, Pred(\chi)), ...,

I_{k,m_k} := Extend(v, Pred(\chi)))(

\hat{\sigma}\psi(\hat{\rho}(I_1, \alpha) \hat{\times} ... \hat{\times} \hat{\rho}(I_k, \alpha)))))

commit_transaction

```

Like the TQuel retrieve statement, this transaction first creates a new temporal relation denoted by I_{k+1} and then assigns to it the historical state represented by the specified algebraic expression. The snapshot state specified in every Quel retrieve statement (a target list and where clause) is equivalent to an algebraic expression that represents cartesian product of the snapshot states associated with tuple variables, followed by selection by the where-clause predicate, and then projection on the attributes in the target list. Similarly, the relation state specified in every TQuel retrieve statement is equivalent to an algebraic expression that represents cartesian product of the referenced relation states, followed by selection by the where-clause predicate, historical derivation as specified by the when and valid clauses, and then projection on the attributes in the target list.

Theorem 5.1 *Every TQuel retrieve statement of the form of 5.2 found on page 104 is equivalent to a transaction in our language of the form*

PROOF. For this proof, assume that execution of the above transaction on database (DS_1, tn) produces the database $(DS_2, tn + 1)$ and execution of the TQuel retrieve statement given in 5.2 on database (DS'_1, tn) produces the database $(DS'_2, tn + 1)$. Also assume that (DS_1, tn) and (DS'_1, tn) are temporally equivalent databases. Then, to prove that the transaction is the algebraic equivalence of the TQuel retrieve statement, we must show that $(DS_2, tn + 1)$ and $(DS'_2, tn + 1)$ are temporally equivalent. From the assumptions that the TQuel retrieve statement is type correct and the databases, before the transaction (or retrieve statement) is executed, are temporally equivalent, it follows that the transaction is also type correct. Hence, to show that the databases are temporally equivalent, we need show only that, immediately following the execution of the transaction on (DS_1, tn) and execution of the TQuel retrieve statement on (DS'_1, tn) ,

$$\text{Findstate}(DS_2(I_{k+1}), tn + 1) = \text{TF}_T(DS'_2(I_{k+1}), tn + 1).$$

It follows from the definitions of the commands `define_relation` and `modify_relation` and the semantic functions P, C, E, and T from Chapter 4 that

$$\text{Findstate}(DS_2(I_{k+1}), tn + 1) =$$

$$\begin{aligned} & \hat{\pi}_{\{(I_{k+1,1}, (I_{i_1, e_1}, I_{i_1, e_1})), \dots, (I_{k+1,n}, (I_{i_n, e_n}, I_{i_n, e_n}))\}}(\\ & \delta_{\Gamma_T, \{(I_{1,1}, \text{Extend}(\Phi_v, \text{Pred}(\Phi_x))), \dots, (I_{k,m_k}, \text{Extend}(\Phi_v, \text{Pred}(\Phi_x)))\}}(\\ & \hat{\sigma}_{\Psi_\alpha}(\hat{\rho}(I_1, \Phi_\alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_k, \Phi_\alpha))). \end{aligned} \quad (5.3)$$

If we let this historical state be R , we must show that $R = TF_T(R'_{k+1}, tn + 1)$, where R'_{k+1} is the TQuel embedded temporal relation denoted by I_{k+1} in DS'_2 . From set theory and the definition of TF_T , it follows that R and $TF_T(R'_{k+1}, tn + 1)$ are equal if, and only if, the following holds.

$$(\forall r, r \in R, \forall I, I \in \mathcal{A}_R, \forall t, t \in Valid(r(N(I))),$$

$$\begin{aligned} & \exists r'_{k+1}, (r'_{k+1} \in R'_{k+1} \\ & \quad \wedge \forall I', I' \in \mathcal{A}_R, Value(r(I')) = r'_{k+1}(I') \\ & \quad \wedge Before(Pred(r'_{k+1}(Start)), tn + 1) \\ & \quad \wedge Before(tn + 1, r'_{k+1}(Stop)) \\ & \quad \wedge t \in Extend(r'_{k+1}(From), Pred(r'_{k+1}(To))) \\ &) \end{aligned} \tag{5.4}$$

)

$$\begin{aligned} & \wedge (\forall r, r \in R, \forall r'_{k+1}, (r'_{k+1} \in R'_{k+1} \\ & \quad \wedge \forall I, I \in \mathcal{A}_R, r'_{k+1}(I) = Value(r(I)) \\ & \quad \wedge Before(Pred(r'_{k+1}(Start)), tn + 1) \\ & \quad \wedge Before(tn + 1, r'_{k+1}(Stop))), \end{aligned}$$

$$\forall I, I \in \mathcal{A}_R,$$

$$Extend(r'_{k+1}(From), Pred(r'_{k+1}(To))) \subseteq Valid(r(I))$$

)

where $\mathcal{A}_R = \{I_{k+1,1}, \dots, I_{k+1,n}\}$. Recall that the first clause ensures that each tuple in R has at least one value-equivalent tuple in R'_{k+1} that was active at transaction $tn + 1$ and the second clause ensures that each subset of value-equivalent tuples in R'_{k+1} , active at transaction $tn + 1$, is represented by a single tuple in R .

To prove the validity of (5.4), we show that the tuple calculus semantics for R , along with the tuple calculus semantics for R'_{k+1} given in (5.2), implies (5.4). First, we construct the tuple calculus statement for R from the definitions of the historical operators \hat{x} , $\hat{\sigma}$, δ , and $\hat{\pi}$, using straightforward substitution, change of variable, and simplification (i.e., the definition of $\hat{\rho}(I_1, \Phi_\alpha) \hat{x} \dots \hat{x} \hat{\rho}(I_k, \Phi_\alpha)$ obtained from the \hat{x} operator is substituted for references to the historical state in the definition of $\hat{\sigma}$, etc.), arriving at (5.5).

$$\begin{aligned}
R = & \hat{\pi}_{\{(I_{k+1,1}, (I_{i_1,a_1}, I_{i_1,a_1})), \dots, (I_{k+1,n}, (I_{i_n,a_n}, I_{i_n,a_n}))\}}(\\
& \delta_{\Gamma_r, \{(I_{1,1}, Extend(\Phi_v, Pred(\Phi_x))), \dots, (I_{k,m_k}, Extend(\Phi_v, Pred(\Phi_x)))\}}(\\
& \hat{\sigma}_{\Psi_\psi}(\hat{\rho}(I_1, \Phi_\alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_k, \Phi_\alpha))) \triangleq
\end{aligned}$$

$$\begin{aligned}
1 & \{r^n \mid (\forall I, I \in \mathcal{A}_R, \forall t, t \in Valid(r(I)), \\
2 & (\exists r_1) \dots (\exists r_k) (\exists IN_1) \dots (\exists IN_k), \\
3 & (r_1 \in \hat{\rho}(I_1, \Phi_\alpha) \wedge \dots \wedge r_k \in \hat{\rho}(I_k, \Phi_\alpha)) \\
4 & \wedge IN_1 \in Interval(Valid(r_1(I_{1,1}))) \wedge \dots \\
5 & \wedge IN_k \in Interval(Valid(r_k(I_{k,1}))) \\
6 & \wedge \forall l, 1 \leq l \leq n, Value(r(I_{k+1,l})) = Value(r_{i_l}(I_{i_l,a_l})) \\
7 & \wedge \Psi_\psi(r_1 \times \dots \times r_k) \\
8 & \wedge \Gamma_r((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)) \\
9 & \wedge t \in Extend(\Phi_v((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)), \\
10 & \quad Pred(\Phi_x((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)))) \\
11 &)) \\
12 & \wedge ((\forall r_1) \dots (\forall r_k) (\forall IN_1) \dots (\forall IN_k) \\
13 & (r_1 \in \hat{\rho}(I_1, \Phi_\alpha) \wedge \dots \wedge r_k \in \hat{\rho}(I_k, \Phi_\alpha)) \\
14 & \wedge IN_1 \in Interval(Valid(r_1(I_{1,1}))) \wedge \dots \\
15 & \wedge IN_k \in Interval(Valid(r_k(I_{k,1}))) \\
16 & \wedge \forall l, 1 \leq l \leq n, Value(r_{i_l}(I_{i_l,a_l})) = Value(r(I_{k+1,l})) \\
17 & \wedge \Psi_\psi(r_1 \times \dots \times r_k) \\
18 & \wedge \Gamma_r((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)) \\
19 &), \\
20 & \forall I, I \in \mathcal{A}_R, \\
21 & Extend(\Phi_v((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)), \\
22 & \quad Pred(\Phi_x((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)))) \subseteq Valid(r(I)) \\
23 &) \\
24 & \wedge (\exists I, I \in \mathcal{A}_r \wedge Valid(r(I)) \neq \emptyset) \\
25 & \}
\end{aligned} \tag{5.5}$$

The three main clauses in the above calculus statement correspond to the three clauses in the definition of π , which appears on page 30. The \times operator contributes the phrase $r_1 \in \hat{\rho}(I_1, \Phi_\alpha) \wedge \dots \wedge r_k \in \hat{\rho}(I_k, \Phi_\alpha)$ that appears in lines 3 and 13 of the calculus statement. The σ operator contributes the predicate found on lines 7 and 17 and the δ operator contributes the predicates found on lines 4-5, 8-10, 14-15, and 18-22.

We now use the definitions and lemmas presented earlier, along with set theory and (5.5), to prove the first clause of (5.4). The first clause in (5.5), along with the definition of $\hat{\rho}$ and the assumption that the databases (DS_1, tn) and (DS'_1, tn) are temporally equivalent implies that

$$\begin{aligned}
& \forall r, r \in R, \forall I, I \in A_r, \forall t, t \in Valid(r(I)), \\
& (\exists r_1) \dots (\exists r_k) (\exists IN_1) \dots (\exists IN_k), \\
& (r_1 \in TF_T(R'_1, \Phi_\alpha) \wedge \dots \wedge r_k \in TF_T(R'_k, \Phi_\alpha)) \\
& \wedge IN_1 \in Interval(Valid(r_1(I_{1,1}))) \wedge \dots \\
& \quad \wedge IN_k \in Interval(Valid(r_k(I_{k,1}))) \\
& \wedge \forall l, 1 \leq l \leq n, Value(r(I_{k+1,l})) = Value(r_{i_l}(I_{l,a_l})) \\
& \wedge \Psi_\psi(r_1 \times \dots \times r_k) \\
& \wedge \Gamma_\tau((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)) \\
& \wedge t \in Extend(\Phi_v((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)), \\
& \quad Pred(\Phi_x((I_{1,1}, IN_1), \dots, (I_{k,1}, IN_k)))) \\
&)
\end{aligned} \tag{5.6}$$

Applying Lemma 5.1 and the definitions of Ψ_ψ and Ψ'_ψ to (5.6) results in

$$\begin{aligned}
& \forall r, r \in R, \forall I, I \in A_r, \forall t, t \in Valid(r(I)), \\
& (\exists r'_1) \dots (\exists r'_k), \\
& (r'_1 \in R'_1 \wedge \dots \wedge r'_k \in R'_k) \\
& \wedge \forall l, 1 \leq l \leq n, Value(r(I_{k+1,l})) = r'_{i_l}(I_{l,a_l}) \\
& \wedge \Psi'_\psi(r'_1(I_{1,1}), \dots, r'_k(I_{k,m_k})) \\
& \wedge \Gamma_\tau((I_{1,1}, Extend(r'_1(From), Pred(r'_1(To)))), \dots, \\
& \quad (I_{k,1}, Extend(r'_k(From), Pred(r'_k(To)))))) \\
& \wedge \forall j, 1 \leq j \leq k, Before(Pred(r'_j(Start)), \Phi_\alpha) \wedge Before(\Phi_\alpha, r'_j(Stop)) \\
& \wedge t \in Extend(\Phi_v((I_{1,1}, Extend(r'_1(From), Pred(r'_1(To)))), \dots, \\
& \quad (I_{k,1}, Extend(r'_k(From), Pred(r'_k(To)))))), \dots, \\
& \quad Pred(\Phi_x((I_{1,1}, Extend(r'_1(From), Pred(r'_1(To)))), \dots, \\
& \quad (I_{k,1}, Extend(r'_k(From), Pred(r'_k(To))))))), \dots, \\
&))
\end{aligned} \tag{5.7}$$

Applying Lemma 5.2 to (5.7) results in

$$\begin{aligned}
 & \forall r, r' \in R, \forall I, I \in \mathcal{A}_R, \forall t, t \in \text{Valid}(r(I)), \\
 & (\exists r'_1) \dots (\exists r'_k), \\
 & (r'_1 \in R'_1 \wedge \dots \wedge r'_k \in R'_k \\
 & \wedge \forall l, 1 \leq l \leq n, \text{Value}(r(I_{k+1,l})) = r'_{l_1}(I_{l_1,a_l}) \\
 & \wedge \Psi'_v(r'_1(I_{1,1}), \dots, r'_k(I_{k,m_k})) \\
 & \wedge \Gamma'_\tau((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To}))) \\
 & \wedge \forall j, 1 \leq j \leq k, \text{Before}(\text{Pred}(r'_j(\text{Start})), \Phi'_\alpha) \wedge \text{Before}(\Phi'_\alpha, r'_j(\text{Stop})) \\
 & \wedge t \in \text{Extend}(\Phi'_v((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To}))), \\
 & \quad \text{Pred}(\Phi'_v((r'_1(\text{From}), r'_1(\text{To})), \dots, (r'_k(\text{From}), r'_k(\text{To})))) \\
 & \quad)) \\
 \end{aligned} \tag{5.8}$$

The third clause of (5.5) on page 110 implies that $\forall r, r \in R, (\exists I)(\exists t), I \in \mathcal{A}_R \wedge t \in \text{Valid}(r(I))$. Hence, applying Lemma 5.3 and the tuple calculus statement for R'_{k+1} in (5.2) on page 105 to (5.8) results in

$$\begin{aligned}
 & \forall r, r \in R, \forall I, I \in \mathcal{A}_R, \forall t, t \in \text{Valid}(r(I)), \\
 & \exists r'_{k+1}, (r'_{k+1} \in R'_{k+1} \\
 & \wedge \forall I, I \in \mathcal{A}_R, \text{Value}(r(I)) = r'_{k+1}(I) \\
 & \wedge \text{Before}(\text{Pred}(r'_{k+1}(\text{Start})), tn + 1) \\
 & \wedge \text{Before}(tn + 1, r'_{k+1}(\text{Stop})) \\
 & \wedge t \in \text{Extend}(r'_{k+1}(\text{From}), \text{Pred}(r'_{k+1}(\text{To}))) \\
 &)
 \end{aligned}$$

Thus, the first clause of (5.4) is shown to hold. A similar argument can be made, starting with the second main clause of (5.5), to show that the second clause of (5.4) holds. Since (5.4) holds, R and $TF_T(R'_{k+1}, tn + 1)$ are equivalent and the transaction is the algebraic equivalence of the indicated TQuel retrieve statement. ■

5.3 TQuel Aggregates

TQuel aggregates [Snodgrass et al. 1987] are a superset of the Quel aggregates. Hence, each of Quel's six non-unique aggregates (i.e., count, any, sum, avg, min, and max) and three unique aggregates (i.e., countU, sumU, and avgU) has a TQuel counterpart. The TQuel

version of each of these aggregates performs the same fundamental operation as its Quel counterpart, with one significant difference. Because a historical relation state represents the changing value of its attributes and aggregates are computed from the entire state, aggregates in TQuel return a distribution of values over time. Hence, while in Quel an aggregate with no by-list returns a single value, in TQuel the same aggregate returns a sequence of values, each assigned its valid times. When there is a by-list, an aggregate in TQuel returns a sequence of values for each value of the attributes in the by-list.

Several aggregates are found only in TQuel: standard deviation (`stdev` and `stdevU`), average time increment (`avgti`), the variability of time spacing (`varts`), oldest value (`first`), newest value (`last`), From-To interval with the earliest From time (`earliest`), and From-To interval with the latest From time (`latest`).

Each TQuel aggregate has a counterpart in our historical algebra. The algebraic equivalences of TQuel aggregates are defined in terms of the historical aggregate functions \widehat{A} and \widehat{AU} , which were defined in Section 3.4. Before defining the algebraic equivalences of TQuel aggregates in the context of a TQuel retrieve statement however, we consider the families of scalar aggregates that appear as parameters to \widehat{A} and \widehat{AU} in the algebraic equivalences of TQuel aggregates. Each aggregate in one of these families of scalar aggregates returns, for a partition of historical state R at time t , the same value returned by its analogous TQuel scalar aggregate for a partition, at time t , of the temporal relation R' 's historical state at the time of transaction tn , where $R = TF_T(R', tn)$.

5.3.1 Aggregate Functions

We define here the families of scalar aggregates that appear as parameters to \widehat{A} and \widehat{AU} in the algebraic equivalences of the TQuel aggregates `count`, `countU`, `first`, and `earliest`. We present these definitions to illustrate our approach for defining the families of scalar aggregates that appear in the algebraic equivalences of TQuel aggregates. The approach can be used to define the families of scalar aggregates found in the algebraic equivalences of the other TQuel aggregates as well. The aggregates `count` and `countU` illustrate how conventional aggregate operators, now applied to historical states, can be handled. The aggregate `first` is an example of an aggregate that evaluates to a non-temporal domain such as character but uses an attribute's valid time in a way different from the conventional aggregate operators. Finally, `earliest` illustrates an aggregate that evaluates to an interval.

For the definitions that follow, let R be a historical state of m -tuples over the relation signature z_R with attributes $\mathcal{A}_R = \{ I_1, \dots, I_m \}$ and Q be a historical state of m -tuples over the relation signature z_Q with attributes \mathcal{A}_Q , where $\mathcal{A}_Q \subseteq \mathcal{A}_R$.

Although the scalar aggregate `Count`, introduced on page 38, is sufficient to define the algebraic equivalence of the TQuel aggregates `count` and `countU` for an aggregation window of length zero (i.e., an instantaneous aggregate), it is not sufficient to define the algebraic equivalence of `count` and `countU` for an aggregation window of any other length. Hence,

we define another family of scalar aggregates Countint_{I_a} , $1 \leq a \leq m$, that accommodates aggregation windows of arbitrary length by counting intervals rather than values.

$$\text{Countint}_{I_a}(q, t, R) = \sum_{r \in R} |\text{Interval}(\text{Valid}(r(I_a)))|$$

where I_a is an attribute of both Q and R , $q \in Q$, and $t \in T$. Recall that Interval , formally defined in Appendix B, returns the set of intervals contained in its argument. Hence, Countint simply sums the number of intervals in the time-stamp of attribute I_a of each tuple in R .

Next, we consider the TQuel aggregate **first**. This aggregate requires a family of scalar aggregate functions Firstvalue_{I_a} , $1 \leq a \leq m$, where Firstvalue_{I_a} produces the oldest value component of attribute I_a . That is,

$$\begin{aligned} \text{Firstvalue}_{I_a}(q, t, R) \in \{u \mid R \neq \emptyset \rightarrow \exists r. (r \in R \wedge \forall r'. r' \in R, \\ \text{First}(r(I_a)) \leq \text{First}(r'(I_a)) \wedge u = \text{Value}(r(I_a)) \\ \}) \\ \wedge R = \emptyset \rightarrow u = \text{Nullvalue}(I_a) \\ \} \end{aligned}$$

where Nullvalue is an auxiliary function that returns a special null value for the domain associated with its argument. Note that the set $\{u \mid \dots\}$ need not be a singleton set. If there are two or more elements in the set, Firstvalue returns only one element, that element being selected arbitrarily. This procedure is the same as that used by the TQuel aggregate **first** to select the oldest value component of an attribute when there are multiple values that satisfy the selection criteria. If R is empty, Firstvalue returns a special null value for the domain associated with attribute I_a .

Finally, we define the algebraic equivalence of the TQuel aggregate **earliest**. Unlike other TQuel aggregates, which produce a distribution of scalar values over time, **earliest** produces a distribution of intervals over time. Defining the algebraic equivalence of this aggregate is slightly more complicated owing to this distinction. We first introduce a family of auxiliary functions OrderInt_{I_a} , $1 \leq a \leq m$, that orders chronologically all distinguishable intervals in the time-stamp of attribute I_a for tuples of historical state R .

$$\begin{aligned}
S \triangleq \text{OrderInt}_{I_a}(R) &\leftarrow (\forall r)(\forall IN), (r \in R \wedge IN \in \text{Interval}(\text{Valid}(r(I_a)))), \\
&\exists v, 1 \leq v \leq |S| \wedge S_v = IN \\
&\wedge \forall v, 1 \leq v \leq |S|, \\
&(\exists r)(\exists IN), (r \in R \wedge IN \in \text{Interval}(\text{Valid}(r(I_a))) \wedge S_v = IN) \\
&\wedge \forall v, 2 \leq v \leq |S|, \\
&(First(S_{v-1}) < First(S_v)) \\
&\vee (First(S_{v-1}) = First(S_v) \wedge Last(S_{v-1}) < Last(S_v)))
\end{aligned}$$

where S is a sequence of length $|S|$ and S_v is the v^{th} element of sequence S . Evaluating $\text{OrderInt}_{I_a}(R)$ results in a sequence of the intervals appearing in the time-stamp of attribute I_a of tuples in R . The intervals are ordered from earliest starting time to latest starting time. When two or more intervals have the same starting time, they are ordered from the earliest stopping time to the latest stopping time. The first clause states that each interval in the time-stamp of attribute I_a of a tuple in R appears in S , the second clause states that no additional intervals are present, and the third clause provides the ordering conditions.

Now, we can define a family of scalar aggregate functions Position_{I_a} , $1 \leq a \leq m$, where Position_{I_a} first identifies, for a tuple q and time t , the interval in the valid-time component of attribute I_a in q that overlaps t and then calculates the position of that interval in $\text{OrderInt}_{I_a}(R)$, for a historical state R . If no interval in the valid-time component of attribute I_a overlaps t or the interval is not in $\text{OrderInt}_{I_a}(R)$, Position_{I_a} returns zero.

$$\begin{aligned}
\text{Position}_{I_a}(q, t, R) = u \rightarrow & ((\exists IN)(\exists S_v), (IN \in \text{Interval}(\text{Valid}(q(I_a)))) \\
&\wedge 1 \leq v \leq |\text{OrderInt}_{I_a}(R)| \\
&\wedge S_v \in \text{OrderInt}_{I_a}(R) \\
&\wedge t \in IN \wedge IN = S_v) \\
& \rightarrow u = v \\
&\wedge ((\forall IN)(\forall S_v), (IN \in \text{Interval}(\text{Valid}(q(I_a)))) \\
&\wedge 1 \leq v \leq |\text{OrderInt}_{I_a}(R)| \\
&\wedge S_v \in \text{OrderInt}_{I_a}(R) \\
&), t \notin IN \vee IN \neq S_v \\
& \rightarrow u = 0
\end{aligned}$$

Note that Position , unlike Countint and Firstvalue , requires parameters q and t , as well as R .

Now assume that we are given a family of scalar aggregate functions Smallest_{I_a} , $1 \leq a \leq m$, where Smallest_{I_a} produces the smallest value component of numerical attribute I_a . That is,

$$\begin{aligned} \text{Smallest}_{I_a}(q, t, R) = u &\leftrightarrow R \neq \emptyset \rightarrow \exists r, (r \in R \\ &\quad \wedge \forall r', r' \in R, \text{Value}(r(I_a)) \leq \text{Value}(r'(I_a)) \\ &\quad \wedge u = \text{Value}(r(I_a)) \\ &\quad) \\ &\quad \wedge R = \emptyset \rightarrow u = 0 \end{aligned}$$

The families of scalar aggregates *Position* and *Smallest* are both needed to define the algebraic equivalence of the TQuel aggregate *earliest* for attribute I_a of relation state R' . First, *Position* is used to assign each interval in the time-stamp of attribute I_a of a tuple in $TF_T(R')$ to an integer representing the interval's relative position in the chronological ordering of intervals. Then, *Smallest* is used to determine, from this assignment of intervals to integers, the times, if any, when each interval was the earliest interval. If we assume an aggregation window function $w(t) = 0$ and an empty set of by-clause attributes, the algebraic equivalence of the TQuel aggregate *earliest* for attribute I_a of relation state R' is

$$\hat{\sigma}_{I_{\text{earliest}_1}=I_{\text{earliest}_2}}(\hat{A}_{\text{Smallest}, 0, I_{\text{earliest}_2}, I_{\text{earliest}_1}, \emptyset}(R_{\text{position}}, R_{\text{position}}) \hat{\times} R_{\text{position}}) \quad (5.9)$$

over the attributes $A_{\text{earliest}} = \{I_{\text{earliest}_1}, I_{\text{earliest}_2}\}$ where

$$R_{\text{position}} = \hat{\sigma}_{I_{\text{earliest}_2} \neq 0}(\hat{A}_{\text{Position}, \infty, I_a, I_{\text{earliest}_2}, \emptyset}(R, R)) \quad (5.10)$$

over the attribute $A_{\text{position}} = \{I_{\text{earliest}_2}\}$.

EXAMPLE. Assume that we are given the historical state S_6 from page 30 over the relation signature *Enrollment* with the attributes *{sname, state}*, duplicated below.

$$\begin{aligned} & \{ \langle ("Phil", \{1, 3, 4\}), ("Kansas", \{1, 2, 3\}) \rangle, \\ & \quad \langle ("Phil", \{1, 3, 4\}), ("Utah", \{4, 5, 6\}) \rangle, \\ & \quad \langle ("Norman", \{1, 2, 5, 6\}), ("Utah", \{1, 2, 5, 6\}) \rangle, \\ & \quad \langle ("Norman", \{1, 2, 5, 6\}), ("Texas", \{7, 8\}) \rangle \} \end{aligned}$$

If we also assume an aggregation window function $w(t) = 0$ and an empty set of by-clause attributes, then *earliest* for attribute *state* of historical state S_6 is

$$\hat{\sigma}_{I_{\text{earliest}_1}} = I_{\text{earliest}_2}(\hat{A}_{\text{Smallest}, 0, I_{\text{earliest}_2}, I_{\text{earliest}_1}, \emptyset}(R_{\text{position}}, R_{\text{position}}) \hat{\times} R_{\text{position}}) = \\ \{ \langle (1, \{1, 2\}), (1, \{1, 2\}) \rangle, \\ \langle (2, \{3\}), (2, \{1, 2, 3\}) \rangle, \\ \langle (3, \{4, 5, 6\}), (3, \{4, 5, 6\}) \rangle, \\ \langle (5, \{7, 8\}), (5, \{7, 8\}) \rangle \}$$

where R_{position} is

$$\hat{\sigma}_{I_{\text{earliest}_2}} \neq \emptyset(\hat{A}_{P_{\text{position}}, \infty, \text{state}, I_{\text{earliest}_2}, \emptyset}(S_6, S_6)) = \\ \{ \langle (1, \{1, 2\}) \rangle, \\ \langle (2, \{1, 2, 3\}) \rangle, \\ \langle (3, \{4, 5, 6\}) \rangle, \\ \langle (4, \{5, 6\}) \rangle, \\ \langle (5, \{7, 8\}) \rangle \}$$

□

As illustrated in this example, the algebraic equivalence of `earliest` is a two-attribute historical state. The valid-time component of the first attribute is the time when the valid-time component of the second attribute was the earliest interval. Also note that the value component of both attributes is the position of the valid-time component of the second attribute in $OrderInt_{I_e}(R)$.

5.3.2 In the Target List

In Section 5.2 we showed the algebraic equivalence of the TQuel retrieve statement without aggregates. We now show the algebraic equivalence of a TQuel retrieve statement with aggregates in its target list. We consider changes to the algebraic expression to support one non-unique aggregate in the target list only; similar changes would be needed for each additional aggregate in the target list.

Once again assume that we are given the TQuel database (d'', tn) containing the k embedded temporal relations R'_1, \dots, R'_k on signatures z'_1, \dots, z'_k that induce, respectively, the attributes,

$$\begin{aligned} A_1 &= \{I_{1,1}, \dots, I_{1,m_1}, \text{From, To, Start, Stop}\} \\ &\dots \\ A_k &= \{I_{k,1}, \dots, I_{k,m_k}, \text{From, To, Start, Stop}\} \end{aligned}$$

where, for notational convenience, we assume that $I_{1,1}, \dots, I_{k,m_k}$ are unique. Also, let

i_1, i_2, \dots, i_n and j_1, j_2, \dots, j_p be integers, not necessarily distinct, in the range 1 to k , indicating the tuple variables (possibly repeated) appearing in the target list and aggregate, respectively;

a_l , $1 \leq l \leq n$, be an integer in the range 1 to m_{i_l} , indicating the attribute names appearing in the target list where $(\forall u)(\forall v)$, $(1 \leq u \leq n \wedge 1 \leq v \leq n \wedge u \neq v \wedge i_u = i_v)$, $a_u \neq a_v$;

c_h , $1 \leq h \leq p$, be an integer in the range 1 to m_{j_h} , indicating the attribute names appearing in the aggregate where $(\forall u)(\forall v)$, $(1 \leq u \leq p \wedge 1 \leq v \leq p \wedge u \neq v \wedge j_u = j_v)$, $c_u \neq c_v$; and

j_1, j_2, \dots, j_p be the distinct integers in j_1, j_2, \dots, j_p where $j_1 = i_1$, indicating the p (non-repeated) tuple variables appearing in the aggregate.

Then, the TQuel retrieve statement with the aggregate f'_1 in the target list has the following syntax

range of r'_1 is I_1

...

range of r'_k is I_k

retrieve into persistent I_{k+1} ($I_{k+1,1} = r'_{i_1}.I_{i_1,a_1}, \dots, I_{k+1,n} = r'_{i_n}.I_{i_n,a_n}$,

$I_{k+1,n+1} = f'_1(r'_{j_1}.I_{j_1,c_1} \text{ by } r'_{j_2}.I_{j_2,c_2}, \dots, r'_{j_p}.I_{j_p,c_p}$

for ω'_1

where ψ'_1

(5.11)

when τ'_1)

valid from v' to χ'

where ψ'

when τ'

where $d'(I_j)$, $1 \leq j \leq k$, denotes the embedded temporal relation R'_j . Again, we assume that the statement is type correct for the database (d', tn) . The statement, when executed on the database, creates a new relation denoted by I_{k+1} , computes a new embedded temporal relation R'_{k+1} with attributes

$$\mathcal{A}_{k+1} = \{I_{k+1,1}, \dots, I_{k+1,n}, I_{k+1,n+1}, \text{From, To, Start, Stop}\}$$

and changes the database state d' to map I_{k+1} onto this new relation. The for clause specifies an aggregation window function for the aggregate f'_1 . ω'_1 contains one or more

keywords that determine, along with the time granularity of R'_1, \dots, R'_k , the length of the aggregation window at each time t . The keyword `each instant` represent the aggregation window function $w(t) = 0$ (i.e., an instantaneous aggregate) and the keyword `ever` represents the aggregation window function $w(t) = \infty$ (i.e., a cumulative aggregate). The length of the aggregation window specified by other keywords (e.g., `each day`, `each week`, `each year`) is a function of the underlying time granularity of the database. For example, if the time granularity is a day, then $\omega' = \text{each week}$ translates to the aggregation window function $w(t) = 6$. Also, the aggregation window function need not be a constant function. For example, if the time granularity is a day, then $\omega' = \text{each month}$ translates to the aggregation window function w , where $w(t) = 31$ if t corresponds to January 31 and $w(t) = 28$ if t corresponds to February 28. We let ω_1 denote in our language the same windowing function denoted by ω'_1 and the time granularity of R'_1, \dots, R'_k in TQuel.

Let u_1, u_2, \dots, u_n be integers, not necessarily distinct, in the range 1 to e where signature z_{i_l} maps attribute I_{i_l, a_l} , and DN maps domain name I_{u_l} , onto value domain D_{u_l} , $1 \leq l \leq n$. Also assume $D_{u_{n+1}}$ is the range of the aggregate f'_1 , where DN maps domain name $I_{u_{n+1}}$ onto $D_{u_{n+1}}$. Then, every TQuel retrieve statement of the form of (5.11) is equivalent to a transaction in our language of the form

```

begin_transaction

define_relation( $I_{k+1}$ , temporal, ( $I_{k+1,1}:I_{u_1}, \dots, I_{k+1,n}:I_{u_n}, I_{k+1,n+1}:I_{u_{n+1}}$ )),
modify_relation( $I_{k+1}$ , *, *,
 $\hat{\pi}(I_{k+1,1} := (I_{i_1, a_1} \oplus I_{i_1, a_1}), \dots, I_{k+1,n} := (I_{i_n, a_n} \oplus I_{i_n, a_n})),$  (5.12)
 $I_{k+1,n+1} := (I_{agg_1,p} \oplus I_{agg_1,p})) ($ 
 $\delta\tau, (I_{j_1,1} := \text{Extend}(v, \text{Pred}(\chi)) \cap I_{j_1,1} \cap \dots \cap I_{j_p,1} \cap I_{agg_1,p}, \dots,$ 
 $I_{agg_1,p} := \text{Extend}(v, \text{Pred}(\chi)) \cap I_{j_1,1} \cap \dots \cap I_{j_p,1} \cap I_{agg_1,p}) ($ 
 $\dot{\sigma}\psi \text{ and } I_{j_1,c_1} = I_{agg_1,1} \text{ and } \dots \text{ and } I_{j_p,c_p} = I_{agg_1,p-1} ($ 
 $\hat{\rho}(I_1, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_k, \alpha) \hat{\times} R_{agg_1})))$ 
commit_transaction

```

where

$$\begin{aligned}
R_{agg_1} &= \widehat{\mathbf{A}} f_1, \omega_1, I_{j_1, c_1}, I_{agg_1, p}, (I_{j_2, c_2}, \dots, I_{j_p, c_p}) (\\
&\quad \hat{\pi}(I_{j_1, c_1}, \dots, I_{j_p, c_p})(\hat{\rho}(I_{j_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \alpha)), \\
&\quad \delta\tau_1, (I_{j_1,1} := I_{j_1,1}, \dots, I_{j_p, m_{j_p}} := I_{j_p, m_{j_p}}) (\\
&\quad \dot{\sigma}\psi_1(\hat{\rho}(I_{j_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \alpha)))
\end{aligned} \tag{5.13}$$

with attributes $A_{agg_1} = \{I_{agg_1,1}, \dots, I_{agg_1,p}\}$, where $\forall u, 1 \leq u \leq p - 1, I_{agg_1,u}$ "renames" $I_{j_{u+1}, c_{u+1}}$ and $I_{agg_1,p}$ is the attribute name associated with the aggregate value. Here we assume that f_1 is the family of scalar aggregates (e.g., *Countint*) corresponding to the family of TQuel aggregates f'_1 (e.g., *count*). The expression denoted by (5.13) applies the where and when predicates to the cartesian product of the relation states associated with tuples variables appearing in the aggregate, and applies the aggregate operator to the result. The expression denoted by the fourth parameter of the *modify_relation* command in (5.12) differs only slightly from expression (5.3) on page 108 for a retrieve statement without aggregates. The expanded selection operator provides the necessary linkage between the attributes in the aggregate's by-list and corresponding attributes in the base relation states. The expanded derivation operator imposes the TQuel restriction that the valid time of tuples in the derived state be the intersection of the valid time specified in the valid clause, the valid times of the tuples in the base relation states participating in the aggregation, and the valid time of the aggregate itself. Of course, if f'_1 is a unique aggregate, then \widehat{AU} should be used instead of \widehat{A} in (5.13).

Three changes to (5.12) are required to handle special cases. First, if a tuple variable $j_u, 1 \leq u \leq p$, does not appear outside the aggregate f'_1 in (5.11), then $I_{j_u,1}$ does not appear in the second subscript of the δ operator. Second, if j_1 appears neither outside the aggregate f'_1 in (5.11) nor in its by clause, then $\rho(I_{j_1}, \alpha)$ does not appear in the sequence of cartesian products. Third, if j_1 does not appear outside the aggregate and there is no by clause, then R_{agg_1} is replaced by

$$\begin{aligned} R_{agg_1} \cup & \langle [\text{historical}, (I_{null} : I_{u_{n+1}}), (I_{null} : \text{Nullvalue}(I_{u_{n+1}}) \text{call})] \\ & \vdash (\pi(I_{null})(\delta \text{true}, (I_{agg_1,p} := I_{agg_1,p}, I_{null} := I_{agg_1,p})) \\ & R_{agg_1} \times [\text{historical}, (I_{null} : I_{u_{n+1}}), (I_{null} : \text{Nullvalue}(I_{u_{n+1}}) \text{call})] \rangle \rangle \end{aligned}$$

where, for notational convenience, we assume that I_{null} simply renames $I_{agg_1,p}$. The first change removes the restriction that the valid time of a tuple in the derived state must intersect the valid time of at least one tuple in the base relation state associated with tuple variable j_u . The second change ensures that, when j_1 appears neither outside the aggregate nor in its by clause, output tuples are produced, even if the historical state denoted by $\rho(I_{j_1}, \alpha)$ is empty. The third change ensures that, when j_1 does not appear outside the aggregate and there is no by clause, a value (possibly a distinguished null value) for the aggregate is specified at each time $t, t \in T$.

5.3.3 In the Inner Where Clause

Aggregates may also appear in the where, when, and valid clauses of a TQuel retrieve statement. We now show the algebraic equivalences of TQuel retrieve statements with aggregates in these clauses, first presenting the algebraic equivalence of a TQuel retrieve statement with an aggregate in an inner where clause. Assume that a TQuel aggregate f'_2

appears in ψ_1 in (5.11) and let

g_1, g_2, \dots, g_y be integers, not necessarily distinct, in the range 1 to k , indicating the (possibly repeated) tuple variables appearing in the nested aggregate where $\forall g_u, 1 \leq u \leq y, \exists j_v, 1 \leq v \leq p, g_u = j_v$;

$b_l, 1 \leq l \leq y$, be an integer in the range 1 to m_{g_l} , indicating the attribute names appearing in the nested aggregate where $(\forall u)(\forall v), (1 \leq u \leq y \wedge 1 \leq v \leq y \wedge u \neq v \wedge g_u = g_v), b_u \neq b_v$; and

$\bar{g}_1, \bar{g}_2, \dots, \bar{g}_y$ be the distinct integers in g_1, g_2, \dots, g_y where $\bar{g}_1 = g_1$, indicating the y (non-repeated) tuple variables in the aggregate.

Then, f'_2 in ψ'_1 has the following syntax

$f'_2(r'_{g_1} \cdot I_{g_1, b_1} \text{ by } r'_{g_2} \cdot I_{g_2, b_2}, \dots, r'_{g_y} \cdot I_{g_y, b_y}$
for ω'_2
where ψ'_2
when τ'_2)

As this TQuel retrieve statement is complicated, containing a nested aggregate with a full complement of by, for, where, and when clauses, we should expect a somewhat complicated algebraic equivalence. When modified to account for f'_2 in ψ'_1 , R_{agg_1} becomes

$$\begin{aligned}
R_{agg_1} = & \hat{\pi}(I_{j_2, c_2}, \dots, I_{j_p, c_p}, I_{agg_1, p})(\\
& \hat{A} f_1, \omega_1, I_{j_1, c_1}, I_{agg_1, p}, (I_{j_2, c_2}, \dots, I_{j_p, c_p}, I_{const})(\\
& \hat{\pi}(I_{j_1, c_1}, \dots, I_{j_p, c_p}, I_{const})(\hat{\rho}(I_{j_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \alpha) \\
& \quad \hat{\times} [\text{historical}, (I_{const} : I_{integer}), (I_{const} : "1" @ all)])), \\
& \hat{\pi}(I_{j_1, 1}, \dots, I_{j_p, m_{j_p}}, I_{const})(\\
& \quad \delta \tau_1, (I_{j_1, 1} := I_{j_1, 1}, \dots, I_{j_p, m_{j_p}} := I_{j_p, m_{j_p}}, I_{agg_2, 1} := I_{agg_2, 1}, \dots, \\
& \quad I_{agg_2, y} := I_{agg_2, y}, I_{const} := I_{const} \cap I_{agg_2, y})(\\
& \quad \hat{\sigma} \psi_1 \text{ and } I_{j_2, b_2} = I_{agg_2, 1} \text{ and } \dots \text{ and } I_{j_y, b_y} = I_{agg_2, y-1}(\\
& \quad \hat{\rho}(I_{j_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \alpha) \hat{\times} R_{agg_2} \hat{\times} \\
& \quad [\text{historical}, (I_{const} : I_{integer}), (I_{const} : "1" @ all)])))))) \\
\end{aligned} \tag{5.14}$$

where the attribute name $I_{agg_1,p}$ again refers to the aggregate produced in \hat{A} by f_1 , the reference to the aggregate f'_2 in ψ'_1 is replaced by a reference to $I_{agg_2,y}$, and

$$\begin{aligned} R_{agg_2} = \hat{A} & f_2, \omega_2, I_{g_1, b_1}, I_{agg_2, y}, (I_{g_2, b_2}, \dots, I_{g_y, b_y}) (\\ & \pi(I_{g_1, b_1}, \dots, I_{g_y, b_y})(\hat{\rho}(I_{g_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{g_y}, \alpha)), \\ & \delta \tau_2, (I_{g_1, 1} := I_{g_1, 1}, \dots, I_{g_y, m_{g_y}} := I_{g_y, m_{g_y}}) (\\ & \hat{\sigma} \psi_2(\hat{\rho}(I_{g_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{g_y}, \alpha))) \end{aligned}$$

over the attributes $A_{agg_2} = \{I_{agg_2, 1}, \dots, I_{agg_2, y}\}$, where $\forall u, 1 \leq u \leq y - 1$, $I_{agg_2,u}$ "renames" $I_{g_{u+1}, b_{u+1}}$, $I_{agg_2,y}$ is the attribute name associated with the aggregate value, and f_2 is the family of scalar aggregates corresponding to the family of TQuel aggregates f'_2 .

The relation state $\{(1, T)\}$ is used simply as a constant relation state containing a single tuple whose value component may be an arbitrary element from an arbitrary domain. Here, we effectively add the attribute I_{const} to $\hat{\rho}(I_{j_1}, \Phi_\alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \Phi_\alpha)$ and then use the attribute as an implicit by-list attribute to restrict tuples in the partition of $\hat{\rho}(I_{j_1}, \Phi_\alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{j_p}, \Phi_\alpha)$ at time t to only those tuples that satisfy the predicate in ψ'_1 involving the aggregate f'_2 at time t .

5.3.4 In the Inner When Clause

Assume now that the aggregate f'_2 appears in τ'_1 in (5.12) rather than in ψ'_1 . The only aggregates that can appear in τ'_1 are earliest and latest. Therefore, if we let R_{agg_2} be the two-attribute algebraic equivalence of f'_2 , then the algebraic equivalence of f'_1 would be the same as that given in (5.14) for an aggregate in the inner where clause, with one exception. The reference to f'_2 in τ'_1 is replaced by a reference to $I_{agg_2, y+1}$, not $I_{agg_2, y}$. The valid-time component of $I_{agg_2, y}$ is the time when the valid-time component of $I_{agg_2, y+1}$ was the oldest interval, hence $I_{agg_2, y+1}$ is used in evaluating τ'_1 .

If we assume that f'_2 is earliest, then R_{agg_2} is

$$\begin{aligned} R_{agg_2} = \hat{\sigma} I_{agg_2, y} = I_{agg_2, y+1} (\\ & \hat{A} \text{Smallest} I_{g_1, b_1}, \omega_2, I_{agg_2, y+1}, I_{agg_2, y}, (I_{g_2, b_2}, \dots, I_{g_y, b_y}) (\\ & \pi(I_{agg_2, y+1}, I_{g_2, b_2}, \dots, I_{g_y, b_y})(R_{position} \hat{\times} \hat{\rho}(I_{g_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{g_y}, \alpha)), \\ & \delta \tau_2 \text{ and } I_{g_1, b_1} = I_{agg_2, y+1}, (I_{agg_2, y+1} := I_{agg_2, y+1}, \\ & I_{g_1, 1} := I_{g_1, 1}, \dots, I_{g_y, m_{g_y}} := I_{g_y, m_{g_y}}) (\\ & \hat{\sigma} \psi_2(R_{position} \hat{\times} \hat{\rho}(I_{g_1}, \alpha) \hat{\times} \dots \hat{\times} \hat{\rho}(I_{g_y}, \alpha))) (\\ & \hat{\times} (R_{position} \dot{\cup} [\text{historical}, (I_{agg_2, y+1} : I_{integer}), (I_{agg_2, y+1} : "0" @ all)]))) \end{aligned} \quad (5.15)$$

over the attributes $A_{agg_2} = \{I_{agg_2,1}, \dots, I_{agg_2,n+1}\}$ where

$$R_{position} = \hat{\sigma} \text{ not } (I_{agg_2,n+1} = 0) \quad (5.16)$$

$$\hat{A} \text{Position, infinity, } I_{g_1,b_1}, I_{agg_2,n+1}, () (\hat{p}(I_{g_1}, \alpha), \hat{p}(I_{g_1}, \alpha)))$$

The expression denoted by (5.15), while structurally equivalent to expression (5.9) on page 116, is considerably more complex because of the presence of by, when, and where clauses in the nested aggregate. The attributes of \hat{A} 's first argument now include the attributes appearing in the by clause and the attributes of \hat{A} 's second argument include the attributes of relation states associated with tuple variables appearing in the aggregate. Also, tuples in the second argument are now required to satisfy the where predicate and, for some interval in the time-stamp of attribute I_{g_1,b_1} , the when predicate. Finally, because TQuel assumes earliest and latest return T for an empty partition of R' , the tuple $((0, T))$ is added to $R_{position}$ so that T will be considered the earliest interval at those times when the partition of \hat{A} 's second argument is empty. Recall that *Smallest*, defined on page 116, returns zero when passed an empty relation state.

5.3.5 In the Outer Where Clause

Assume that the TQuel aggregate f'_1 appears in ψ' in (5.11) rather than in the target list. Then, the algebraic equivalence of the TQuel retrieve statement is

```

begin_transaction
define_relation(I_{k+1}, temporal, (I_{k+1,1}:I_{u_1}, ..., I_{k+1,n}:I_{u_n}, I_{k+1,n+1}:I_{u_{n+1}})),
modify_relation(I_{k+1}, *, *,
    π(I_{k+1,1} := (I_{i_1,a_1} ⊕ I_{i_1,a_1}), ..., I_{k+1,n} := (I_{i_n,a_n} ⊕ I_{i_n,a_n}))(
        δτ, (I_{1,1} := Extend(v, Pred(χ)) ∩ I_{j_1,1} ∩ ... ∩ I_{j_p,1} ∩ I_{agg_1,p}, ...
            I_{agg_1,p} := Extend(v, Pred(χ)) ∩ I_{j_1,1} ∩ ... ∩ I_{j_p,1} ∩ I_{agg_1,p})(
                σψ and I_{j_2,c_2} = I_{agg_1,1} and ... and I_{j_p,c_p} = I_{agg_1,p-1}(
                    ρ(I_1, α) × ... × ρ(I_k, α) × R_{agg_1})))))
commit_transaction

```

where the reference to f'_1 in ψ' is replaced by a reference to $I_{agg_1,p}$. Note that the only other change from (5.12) is the elimination of attribute $I_{agg_1,p}$ from the projection, since the aggregate does not appear in the target list.

5.3.6 In the Outer When Clause

Assume now that the aggregate f'_1 appears in τ' in (5.11). Then, the algebraic equivalence of the TQuel retrieve statement is

```

begin_transaction

define_relation( $I_{k+1}$ , temporal ,  $(I_{k+1,1}:I_{u_1}, \dots, I_{k+1,n}:I_{u_n}, I_{k+1,n+1}:I_{u_{n+1}})$ ),
modify_relation( $I_{k+1}$ , *, *,

 $\hat{\pi}(I_{k+1,1} := (I_{i_1,a_1} @ I_{i_1,a_1}), \dots, I_{k+1,n} := (I_{i_n,a_n} @ I_{i_n,a_n})) ($ 
 $\delta\tau, (I_{1,1} := \text{Extend}(v, \text{Pred}(\chi)) \cap I_{j_1,1} \cap \dots \cap I_{j_p,1} \cap I_{agg_1,p}, \dots,$ 
 $I_{agg_1,p} := \text{Extend}(v, \text{Pred}(\chi)) \cap I_{j_1,1} \cap \dots \cap I_{j_p,1} \cap I_{agg_1,p}) ($ 
 $\delta\psi \text{ and } I_{j_2,c_2} = I_{agg_1,1} \text{ and } \dots \text{ and } I_{j_p,c_p} = I_{agg_1,p-1} ($ 
 $\hat{\rho}(I_1, \alpha) \times \dots \times \hat{\rho}(I_k, \alpha) \times R_{agg_1})))$ 

commit_transaction

```

where the reference to f'_1 in τ is replaced by a reference to $I_{agg_1,p+1}$. If the aggregate f'_1 is in v or χ rather than τ , analogous changes would be required.

5.3.7 Multiply-nested Aggregation

The approach described above for handling aggregates in the inner where and when clauses can be used to handle aggregates in a qualifying where or when clause of an aggregate in the outer where, when, or valid clauses. This method of converting TQuel aggregates to their algebraic equivalences, when there is an aggregate in a qualifying clause, can also handle an arbitrary level of nesting of aggregates.

5.3.8 Correspondence Theorem

Now that all possible locations for aggregates in a TQuel retrieve statement have been examined, we can assert that

Theorem 5.2 *Every TQuel retrieve statement has an equivalent transaction in our language.*

PROOF. Induct on the number of aggregates appearing in the statement to arrive at an equivalent algebraic expression, applying the replacements discussed above in Sections 5.3.2 through 5.3.6, as appropriate. Construct a tuple calculus expression for the retrieve statement and the algebraic expression, then prove equivalence using the technique used in the

proof of Theorem 2. While the proof is aided by the presence of auxiliary relation states in the tuple calculus semantics for aggregates [Snodgrass 1987], it is still cumbersome and offers little additional insight. ■

5.4 TQuel Modification Statements

Having shown the algebraic equivalence of the TQuel retrieve statement, both with and without aggregates, we now show the equivalent transaction in our language for each of the TQuel modification statements.

5.4.1 Create Statement

The TQuel create statement, like its Quel counterpart, defines a new relation and provides a signature and class for that relation. Keywords are used to specify the relation's class. If the keyword **persistent** is used, the relation is either a rollback or temporal relation. If the keyword **interval** or **event** is used, the relation is either a historical or temporal relation. If none of these keywords is used, the relation is a conventional snapshot relation. We show here the syntax for a TQuel statement that creates a temporal relation and the statement's corresponding transaction in our language. Transactions for the other forms of the TQuel create statement can be constructed in a similar fashion.

Let u_1, u_2, \dots, u_n be integers, not necessarily distinct, in the range 1 to e where DN maps domain name I_{u_i} onto value domain \mathcal{D}_{u_i} , $1 \leq i \leq n$. Then, the TQuel create statement for an interval-based temporal relation has the following syntax.

```
create persistent interval I(I1 = Iu1, ..., In = Iun)
```

As before, we assume the type correctness of this statement for the TQuel database (d', tn) . The statement, when executed on (d', tn) , creates a new, empty relation denoted by I with attributes

$$\mathcal{A} = \{I_1, \dots, I_n, \text{From}, \text{To}, \text{Start}, \text{Stop}\}$$

and changes the database state d' to map I onto this new relation. The statement's algebraic equivalence is the following transaction.

```
begin_transaction
define_relation(I, temporal, (I1:Iu1, ..., In:Iun))
commit_transaction
```

If we let (d, tn) be the algebraic temporal equivalent of (d', tn) , this transaction, when executed on (d, tn) , simply changes the database state d to make $d(I)$ an empty temporal relation with attributes $\{I_1, \dots, I_n\}$ at transaction tn .

5.4.2 Append Statement

The TQuel append statement creates a new state for a relation by adding tuples to that relation's current state. For the append statement (and the delete and replace statements which follow), assume, as we did for the retrieve statement, that we are given the TQuel database (d', tn) containing the k embedded temporal relations R'_1, \dots, R'_k on signatures z'_1, \dots, z'_k that induce, respectively, the attributes,

$$\mathcal{A}_1 = \{I_{1,1}, \dots, I_{1,m_1}, \text{From, To, Start, Stop}\}$$

...

$$\mathcal{A}_k = \{I_{k,1}, \dots, I_{k,m_k}, \text{From, To, Start, Stop}\}$$

For notational convenience, assume that $I_{1,1}, \dots, I_{k,m_k}$ are unique. Also, assume that (d', tn) contains the embedded temporal relation R'_{k+1} , not necessarily distinct from R'_1, \dots, R'_k , with attributes

$$\mathcal{A}_{k+1} = \{I_{k+1,1}, \dots, I_{k+1,n}, \text{From, To, Start, Stop}\}$$

Furthermore, let i_1, i_2, \dots, i_n be integers, not necessarily distinct, in the range 1 to k and a_l , $1 \leq l \leq n$, be a distinct integer in the range 1 to m_{i_l} . Then, the TQuel append statement has the following syntax

range of r'_1 is I_1

...

range of r'_k is I_k

append to I_{k+1} ($I_{k+1,1} = r'_{i_1}.I_{i_1,a_1}, \dots, I_{k+1,n} = r'_{i_n}.I_{i_n,a_n}$)

valid from v' to x'

where ψ'

when τ'

where $d'(I_j)$, $1 \leq j \leq k+1$, denotes the embedded temporal relation R'_j . Note that, unlike the retrieve statement, no as-of clause is specified. TQuel assumes that changes are always made to a relation's current state.

Every TQuel append statement of this form is equivalent to a transaction in our language of the following form.

```

begin_transaction

modify_relation( $I_{k+1}, *, *, I_{k+1} \cup (\pi(I_{k+1,1} := (I_{1,a_1} \oplus I_{1,a_1}), \dots,$ 
 $I_{k+1,n} := (I_{n,a_n} \oplus I_{n,a_n})) ($ 
 $\delta\tau, (I_{1,1} := \text{Extend}(v, \text{Pred}(\chi)), \dots,$ 
 $I_{k,m_k} := \text{Extend}(v, \text{Pred}(\chi))) ($ 
 $\sigma\psi(I_1 \times \dots \times I_k))))$ 

commit_transaction

```

where $d(I_j)$, $1 \leq j \leq k + 1$, denotes the temporal relation R_j in (d, tn) . The transaction, when executed on (d, tn) , first computes the tuples to be appended to relation R_{k+1} , then does a historical union of R_{k+1} 's current state and those tuples to produce a new relation state, and finally appends this new relation state to R_{k+1} 's state sequence. The expression used to compute the new tuples is structurally the same as expression (5.3) for a retrieve statement, with one exception: it doesn't include any rollback operators because a tuple variable in a TQuel modification statement always references a relation's current state.

5.4.3 Delete Statement

The TQuel delete statement creates a new state for a relation by removing tuples, or portions of tuples, from that relation's current state. It has the following syntax.

```

range of  $r'_1$  is  $I_1$ 
...
range of  $r'_k$  is  $I_k$ 
range of  $r'_{k+1}$  is  $I_{k+1}$ 
delete  $r'_{k+1}$ 
    valid from  $v'$  to  $\chi'$ 
    where  $\psi'$ 
    when  $\tau'$ 

```

Every TQuel delete statement of this form is equivalent to a transaction in our language of the following form.

```

begin_transaction

modify_relation( $I_{k+1}, *, *, I_{k+1} - (\pi(I_{k+1,1}, \dots, I_{k+1,n}) ($ 
 $\delta\tau, (I_{1,1} := I_{1,1} \cap \text{Extend}(v, \text{Pred}(\chi)), \dots,$ 
 $I_{k+1,n} := I_{k+1,n} \cap \text{Extend}(v, \text{Pred}(\chi))) ($ 
 $\delta\psi(I_1 \hat{\times} \dots \hat{\times} I_k \hat{\times} I_{k+1}))))$ 

commit_transaction

```

This transaction, when executed on (d, tn) , first computes the temporal portions of tuples in R_{k+1} that are to be deleted, does a historical difference of R_{k+1} 's current state and those tuple portions to produce a new relation state, and then appends this new relation state to R_{k+1} 's state sequence. The expression used to compute the tuple portions to be deleted differs considerably from the expression for an append statement. R_{k+1} 's current state appears in the sequence of cartesian products, only attributes of R_{k+1} appear in the projection, and the valid times of attributes in each output tuple are required to overlay the valid times of attributes in the tuple's value-equivalent counterpart in R_{k+1} 's current state.

5.4.4 Replace Statement

The TQuel replace statement creates a new state for a relation by first removing tuples, or portions of tuples, from that relation's current state and then adding tuples to the resulting state. It has the following syntax.

```

range of  $r'_1$  is  $I_1$ 
...
range of  $r'_k$  is  $I_k$ 
replace  $r'_{k+1} (I_{k+1,1} = r'_{i_1}, I_{i_1, a_1}, \dots, I_{k+1,n} = r'_{i_n}, I_{i_n, a_n})$ 
    valid from  $v'$  to  $x'$ 
    where  $\psi'$ 
    when  $\tau'$ 

```

Every TQuel replace statement of this form is equivalent to a transaction in our language of the following form.

```

begin_transaction

modify_relation( $I_{k+1}, *, *, (I_{k+1} \leftarrow (\dot{\pi}(I_{k+1,1}, \dots, I_{k+1,n}) ($ 
 $\delta\tau, (I_{1,1} := I_{1,1} \cap \text{Extend}(v, \text{Pred}(\chi)), \dots,$ 
 $I_{k+1,n} := I_{k+1,n} \cap \text{Extend}(v, \text{Pred}(\chi))) ($ 
 $\dot{\sigma}\psi(I_1 \hat{\times} \dots \hat{\times} I_k \hat{\times} I_{k+1}))))))$ 
 $\cup (\dot{\pi}(I_{k+1,1} := (I_{i_1,a_1} @ I_{i_1,a_1}), \dots,$ 
 $I_{k+1,n} := (I_{i_n,a_n} @ I_{i_n,a_n})) ($ 
 $\delta\tau, (I_{1,1} := \text{Extend}(v, \text{Pred}(\chi)), \dots,$ 
 $I_{k,m_k} := \text{Extend}(v, \text{Pred}(\chi))) ($ 
 $\dot{\sigma}\psi(I_1 \hat{\times} \dots \hat{\times} I_k))))))$ 

commit_transaction

```

The transaction, when executed on (d, tn) , first computes the temporal portions of tuples in R_{k+1} that are to be deleted and does a historical difference of R_{k+1} 's current state and those tuple portions to produce a new relation state. It then computes the new tuples to be added and does a historical union of the relation state produced by the difference operator and those tuples. Finally, it appends the resulting state to R_{k+1} 's state sequence. The transaction differs from a delete operation followed by an append operation, however, because both the tuples to be deleted and the tuples to be added are computed from the relation states current at the start of the transaction.

5.4.5 Destroy Statement

The TQuel destroy statement deletes a relation from the database. If the relation is a snapshot or historical relation, it is physically deleted. If, however, the relation is a rollback or temporal relation, it is only logically deleted. Rollback and temporal relations are persistent and remain accessible via rollback operations, even after they are deleted.

```

range of  $r'_1$  is  $I_1$ 
destroy  $r'_1$ 

```

Every TQuel destroy statement of this form is equivalent to a transaction in our language of the following form.

```

begin_transaction
destroy I1
commit_transaction

```

The *destroy* command, like the TQuel *destroy* statement, simply deletes the relation denoted by I_1 from the database. Snapshot and historical relations are physically deleted, while rollback and temporal relations are only logically deleted.

5.4.6 Correspondence Theorem

Now that all TQuel modification statements have been examined, we can assert that

Theorem 5.3 *Every TQuel modification statement has an equivalent transaction in our language.*

PROOF. Construct a tuple calculus expression for each TQuel modification statement and its corresponding algebraic expression. Then prove equivalence using the technique used in the proof of Theorem 2. While the proof is straightforward, it is cumbersome and offers little additional insight. ■

5.5 Language Correspondence

Theorem 5.4 *Our language for database query and update, defined in Chapters 3 and 4, has the expressive power of TQuel.*

PROOF. This theorem follows directly from the correspondence theorems presented in Sections 5.3.8 and 5.4.6. ■

Theorem 5.5 *Our language for database query and update is strictly more powerful than TQuel.*

PROOF. The previous theorem shows that the expressive power of our language is as great as that of TQuel. Now, for two historical relation states R_1 and R_2 , consider the algebraic expression $R_1 \hat{\times} R_2$. Because the semantics of TQuel requires that tuples rather than attributes be time-stamped, this algebraic expression has no counterpart in TQuel. Hence, our language is strictly more powerful than TQuel. ■

5.6 Summary

In this chapter we have shown that our language for database query and update has the expressive power of the temporal query language TQuel. We have given, for each TQuel statement, the transaction that is its algebraic equivalent. We first considered the basic TQuel retrieve statement without aggregates and then more complex TQuel retrieve statements with aggregates in their target lists, where clauses, and when clauses. Finally we considered the create, append, delete, replace, and destroy modification statements. Hence, we have shown that the language is sufficient in expressive power to serve as the underlying evaluation mechanism for TQuel.

In the next chapter we extend the language defined in Chapters 3 and 4 to accommodate views.

Chapter 6

Adding Support for Views

A *base relation* is an autonomous, named relation stored in the database [Date 1986B]. It is autonomous in that it is not defined in terms of other relations. In contrast, a *view* is a named relation that is defined in terms of other named relations, either base relations or other views [Chamberlin et al. 1975, Date 1986B]. A *view definition* is simply the algebraic expression that defines the scheme and state of a view. Whereas base relations are stored in the database, views may, but need not, be stored in the database. We illustrate the relationship between views and base relations by a simple example.

EXAMPLE. Let S denote a snapshot relation whose current signature specifies the attributes {*sname*, *course*} and whose current state is

$$\{ \langle "Phil", "English" \rangle, \langle "Norman", "English" \rangle, \langle "Norman", "Math" \rangle \}.$$

Now consider the three views SP , SM , and SU , each defined by the command `define_view`, whose arguments are the identifier that names the view and the expression that defines the view.

```
define_view(SP, σsname="Phil"(S))
define_view(SM, σsname="Marilyn"(S))
define_view(SU, π(sname)(SP ∪ SM))
```

SP and SM are views, defined in terms of the snapshot relation S . Their signatures, like that of S , specify the attributes {*sname*, *course*}. SP 's state contains the single tuple $\langle "Phil", "English" \rangle$ and SM 's state is empty. SU is also a view, but has only the attribute *sname* in its signature. SU 's state contains the single tuple $\langle "Phil" \rangle$. SP and SM , because they are defined in terms of S , depend on S . Similarly, SU , because it is defined in terms of SP and SM , depends indirectly on S . The view dependency graph for S is given in Figure 6.1. \square

Base relations, because they are autonomous, change only when transactions containing commands that explicitly name the relations are executed. In contrast, views, because

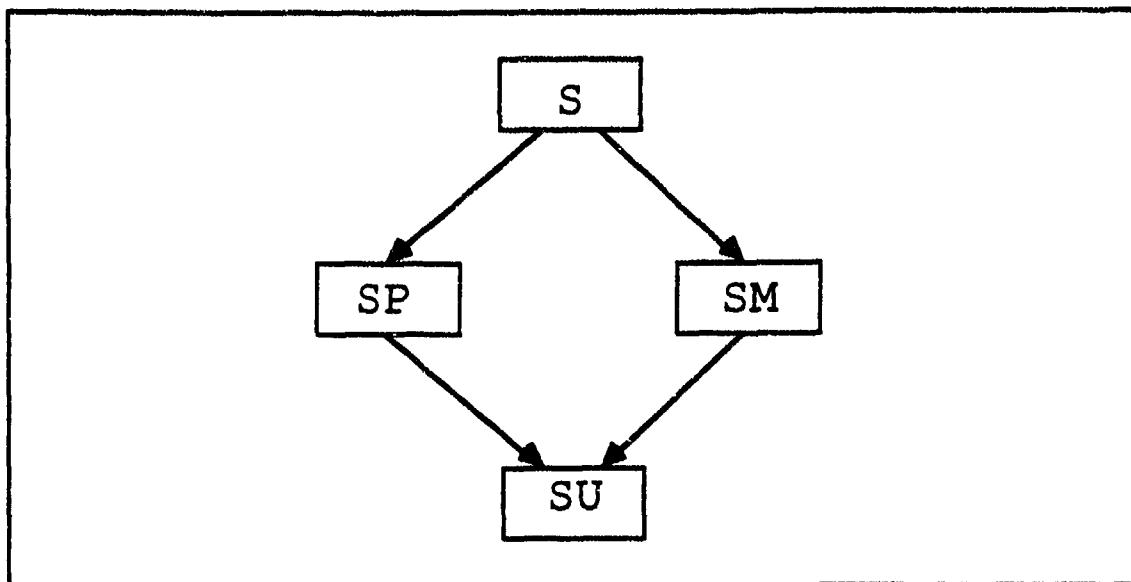


Figure 6.1: View Dependency Graph for Base Relation S

they are functions of other relations, change whenever one of their underlying relations changes. For example, a change to either the scheme or state of S would propagate to views SP, SM, and SU. Views have several advantages. They simplify the users' perceptions of the database, allow users to see the same data differently, and provide security by hiding data from users [Date 1986D]. They also can be used to represent stored recurring queries. In Chapters 3 and 4 we defined an algebraic language for query and update of temporal databases containing base relations only. In this chapter we extend our language to accommodate views as well as base relations. We consider the problem of maintaining views in a temporal database in which both the scheme and state of base relations are allowed to change over time. The problem of updating databases through views (i.e., mapping user-specified updates to views onto updates to the views' underlying relations) is not considered; this problem has already been studied extensively, with generally discouraging results [Bancilhon & Spyros 1981, Cosmadakis & Papadimitriou 1984, Furtado et al. 1979, Furtado & Casanova 1985, Keller 1985, Keller 1986].

6.1 Background

There are two basic strategies for maintaining a view. One strategy, which is the traditional way of maintaining a view, is to store only the view definition in the database and to use query modification to convert queries against the view into queries against the view's underlying base relations [Stonebraker 1975]. In this strategy, a query that contains reference(s) to a view is syntactically augmented before being evaluated; each reference to the view is replaced with the view's definition, and the resulting query, which contains only references to base relations, is optimized and then evaluated.

EXAMPLE. Under query modification, the expression

$$\sigma_{course="English"}(S^r)$$

would be converted to the equivalent expression

$$\sigma_{sname="Phil"} \text{ and } course="English"(S)$$

and then evaluated. □

Query modification requires only that view definitions be stored in the database. Because neither the class, signature, nor state of a view is ever computed and stored in the database, views maintained using this strategy are referred to as *unmaterialized* views. A variation of this strategy, which we refer to as *in-line view evaluation*, is simply to evaluate, whenever a query is executed, the definition of each view named in the query and to treat the resulting views as constant relation states in the query, without storing them in the database.

The other strategy for maintaining a view is to store the class, signature, and state of the view, along with its definition, in the database and to treat queries against the view identically to queries against a base relation. Views maintained using this strategy are referred to as *materialized* views. The strategy has several variations, each characterized by when and how a view is updated to reflect changes to its underlying relations. A materialized view can be updated any time after a change to one of its underlying relations as long as its type (i.e., class and signature) and state are consistent with the type and state of each of its underlying relations whenever it is accessed during query evaluation. There is a spectrum of update strategies that satisfy this criterion, the possible strategies being bounded by update immediately after each change to an underlying relation and by update, if required, just before an access during query evaluation. These strategies are referred to, respectively, as *immediate view materialization* and *deferred view materialization* [Hanson 1987A, Roussopoulos 1987]. Orthogonally, *recomputed view materialization* refers to a strategy in which a view is updated by recomputing the entire view while *incremental view materialization* refers to a strategy in which a view is updated using a differential update algorithm [Blakeley et al. 1986A, Hanson 1987A, Hanson 1987B, Horwitz 1985, Horwitz & Teitelbaum 1986]. Hence, four of the strategies for maintaining materialized views are immediate-recomputed, immediate-incremental, deferred-recomputed, and deferred-incremental.

Figure 6.2 classifies database relations by type and view maintenance strategy. We assume that base relations, unlike views, are always materialized and that updates to base relations are always immediate, never deferred. It is important to note that the presence of views and the choice of a view maintenance strategy can affect the performance, but not the results, of query processing. Execution of a query that references a view always produces the same result as execution of its equivalent query after query modification, independent of the strategy used to maintain the view.

Although to our knowledge there has been no previous work on maintenance of views in temporal databases, there has been considerable research applicable to incremental materialization of views in snapshot databases. Incremental view materialization brings a

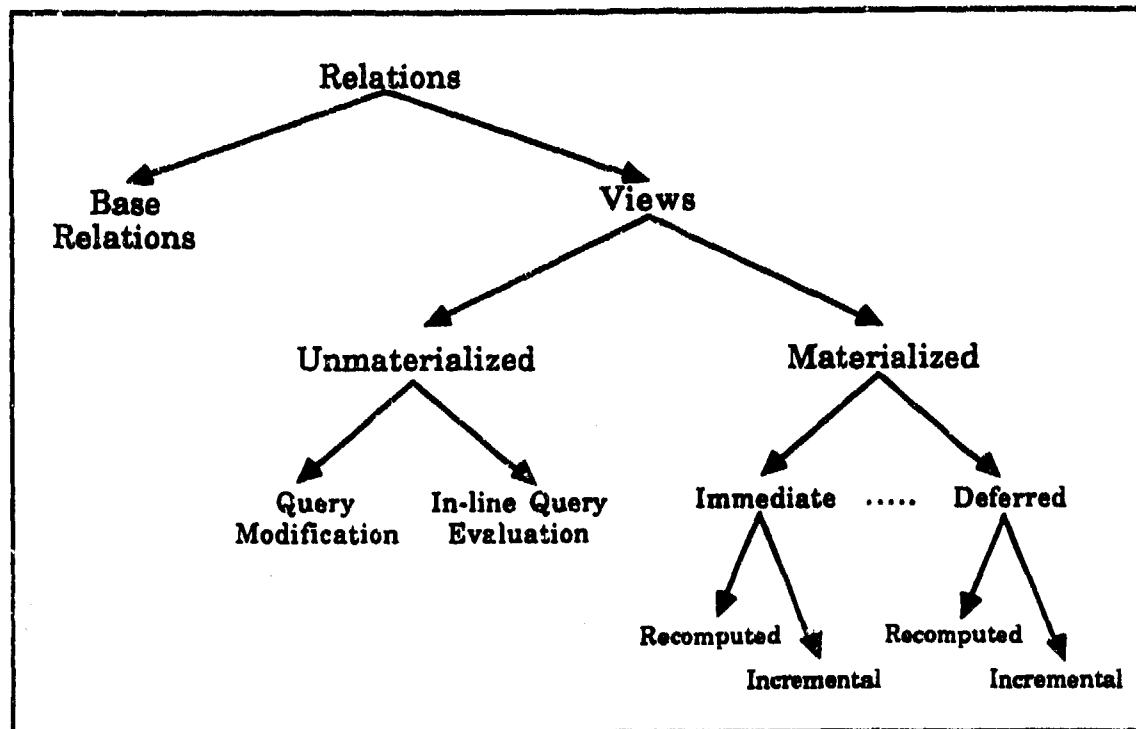


Figure 6.2: Classification of Relations by Type and View Maintenance Strategy

view up-to-date following the update of one of its underlying relations by identifying the changes that must be made to the view's old state for the view's new state to be consistent with the new states of its underlying relations, without having to recompute the view itself [Blakeley et al. 1986A]. The changes an update operation makes to a stored relation, either a base relation or a materialized view, are referred to as a *differential*. Severance and Lohman have discussed the application of differential files to the maintenance of large databases [Severance & Lohman 1976], and Woodfill and Stonebraker have proposed that hypothetical relations be implemented using differential files [Woodfill & Stonebraker 1983]. Koenig and Paige have applied the transformational techniques of finite differencing to the automatic maintenance of derived data in the context of a function/binary association data model [Koenig & Paige 1981]. Shmueli and Itai have proposed a structure for incrementally maintaining materialized views in acyclic databases where views are restricted to the projection of attributes over the natural join of all relations in the database [Shmueli & Itai 1984]. Sufficient and necessary conditions for detecting updates to base relations that cannot affect views have been identified [Blakeley et al. 1986B] and incremental versions of the snapshot algebra have been defined [Blakeley et al. 1986A, Horwitz 1985, Horwitz & Teitelbaum 1986].

Hanson has compared the efficiency of several strategies for maintaining views in snapshot databases [Hanson 1987A, Hanson 1988]. His work shows that the efficiency of view maintenance depends heavily on the database processing environment and that no single strategy is always the most efficient. If database operations are predominately

updates, query modification is shown to be more efficient than other view maintenance strategies, primarily because the performance of incremental view materialization degrades severely as the percentage of operations that are updates increases. Incremental view materialization, however, is shown to be more efficient than either query modification or recomputed view materialization if five conditions are satisfied simultaneously: (1) the number of queries against a view is sufficiently higher than the number of updates to its underlying relations, (2) the sizes of the underlying relations are sufficiently large, (3) the selectivity factor of the view predicate is sufficiently low, (4) the percentage of the view retrieved by queries is sufficiently high, and (5) the volatility of the underlying relations, defined as the percentage of tuples that change between accesses to the view, is sufficiently low. Also, Roussopoulos has shown that incremental view materialization is more efficient than recomputed view materialization, and sometimes significantly so, under similar conditions [Roussopoulos 1987].

Differentials and incremental update also have been shown to have application in the maintenance of *snapshots*, another form of derived relation similar to, but distinct from, views [Adiba & Lindsay 1980]. Snapshots are base relations that are derived from other base relations. Unlike views, which are dynamic and change with each change to their underlying relations, snapshots are static and only change, once defined, when they are refreshed. Incremental update using differentials has been shown to be more efficient than expression re-evaluation as a snapshot refresh strategy when the update activity between refreshes is low and the percentage of the base relations retrieved into the snapshot is high [Lindsay et al. 1986]. These conditions are analogous to those under which incremental view materialization is the preferred view maintenance strategy. In the context of support for differential snapshot refresh, Kahler and Risnes have proposed two methods, *sequential logging* and *condensed logging*, for maintaining a base relation's differential. In sequential logging, a relation's differential is simply a sequentially ordered log of all changes to the relation since the last refresh. In condensed logging, a relation's differential is a set of pairs, each pair containing, for a tuple that has undergone a *net* change since the last refresh, the tuple's image just before the last refresh and the tuple's image after the last update [Kahler & Risnes 1987].

6.2 Approach

Because no view maintenance strategy is the most efficient strategy for all database processing environments, our goal in this chapter is to extend our language sufficiently to support both unmaterialized and materialized views. We extend the language to accommodate query modification and in-line view evaluation when views are unmaterialized and the immediate-recomputed and immediate-incremental strategies when views are materialized. The additional changes that would be required to accommodate deferred view materialization are straightforward and are discussed informally in the next chapter. New commands are needed to define views and to specify view maintenance strategies. Also, the semantics of existing commands must be extended to account for the presence of views in

the database. Furthermore, because the language allows the scheme, as well as the state, of a base relation to change over time, existing commands must be redefined to allow only changes to a base relation's scheme consistent with the definition of all views that depend on the relation. Finally, incremental versions of the snapshot and historical algebras, defined in terms of relation states and differentials rather than just relation states, are needed to support incremental view materialization.

We emphasize support for incremental materialization of views because this strategy likely will be applicable to an even larger subclass of views in temporal databases than in snapshot databases. The probability that incremental view materialization is the preferred strategy is unchanged if a view is defined as a function of the current state of a snapshot or rollback relation. The probability, however, is higher, and possibly substantially higher, if the view is defined as a function of the current state of a historical or temporal relation or the past state of a rollback or temporal relation. The cost of evaluating an algebraic expression involving historical states will be greater than the cost of evaluating the expression's snapshot analogue because additional processing will be required to handle valid time. Also, the current state of a historical or temporal relation, because it models objects over time rather than at one instant, typically will be larger than its analogous snapshot state. Similarly, because information about an object's past is less likely to be changed, once recorded, than information about the object's present, the current state of a historical or temporal relation typically will be less volatile than its snapshot counterpart. Hence, the current state of a historical or temporal relation typically will be both larger and less volatile than its snapshot counterpart. Furthermore, past states of rollback and temporal relations experience no volatility as they can never be changed. These conditions, large size and low volatility, are exactly the conditions under which incremental materialization is the preferred view maintenance strategy. Finally, incremental view materialization likely will be most applicable to views that denote stored, recurring historical queries, because all tuples in the view would be retrieved on each view access (i.e., execution of the stored query).

In the next section, we use existing algorithms for the incremental maintenance of views in snapshot databases [Blakeley et al. 1986A, Hanson 1987A, Horwitz 1985, Horwitz & Teitelbaum 1986] in defining an incremental version of the snapshot algebra. We then adapt these same algorithms to the incremental update of historical views in defining an incremental version of our historical algebra. After defining incremental versions of the snapshot and historical algebras, we extend the language defined in Chapter 4 to accommodate views. We add three new commands to the language's syntax, redefine the semantic functions for type checking and expression evaluation to account for views, define the semantics of the new commands, and extend the semantics of all existing commands to account for views. We conclude the chapter with a discussion of the restrictions the presence of views has on scheme evolution of base relations.

6.3 Incremental Snapshot Algebra

For materialized views of snapshot relations, incremental view materialization brings the view up-to-date following the update of one of its underlying relations by identifying the tuples that must be inserted into, and the tuples that must be deleted from, the view's old state for the view's new state to be consistent with the new states of its underlying relations, without having to recompute the view itself [Blakeley et al. 1986A]. The net changes (i.e., tuples inserted and tuples deleted) that an update operation makes to a stored relation, either a base relation or a materialized view, is the relation's *differential*. To support incremental view materialization, we need to be able to map the old states of a view's underlying relations and their differentials for an update operation onto the view's corresponding differential for the same update operation. The conventional snapshot algebra, however, does not support this capability, as snapshot operators map either one or two relation states onto a relation state. Hence, in this section we define an incremental version of the snapshot algebra in which each operator is defined as a mapping from either one relation state and its differential or two relation states and their differentials onto a resulting relation state and its corresponding differential.

We first define the function *S_Differential* that computes a snapshot differential and the function *S_Update* that maps a snapshot relation's state just before an update and its differential for that update onto its state immediately after the update. Then, we define an incremental version of the five operators that serve to define the snapshot algebra.

6.3.1 Snapshot Differential

We define the differential for an update operation on a snapshot relation as the set of ordered pairs that records the *before* and *after* images of all tuples that the update changes. Assume that we are given the snapshot relation R . Let R_b be R 's state just before an update and R_a be R 's state immediately after the update. (Throughout this chapter, we use the subscript "b" to denote "before" and "a" to denote "after.") We can define the differential Δ_R for the update in terms of the function *S_Differential* as follows.

$$\begin{aligned} S_{\text{Differential}} : [\text{SNAPSHOT STATE} \times \text{SNAPSHOT STATE}] &\rightarrow \\ &\quad \text{SNAPSHOT DIFFERENTIAL} \end{aligned}$$

$$\begin{aligned} \Delta_R \triangleq S_{\text{Differential}}(R_b, R_a) = \{ (r_b, r_a) \mid & (r_b = \text{NIL} \wedge r_a \in R_a - R_b) \\ & \vee (r_b \in R_b - R_a \wedge r_a = \text{NIL}) \} \end{aligned}$$

Δ_R denotes the changes to R_b that produce R_a . Tuple insertion is denoted by a pair whose first component is NIL and whose second component is the inserted tuple. Tuple deletion is denoted by a pair whose first component is the deleted tuple and whose second component

is NIL. Tuple replacement is denoted by a pair denoting tuple deletion and a pair denoting tuple insertion, each pair containing exactly one component whose value is NIL. Also, a tuple appears as a component of at most one pair. Hence, Δ_R denotes the net changes to R_b .

EXAMPLE. Let S , as in the example on page 132, denote a snapshot relation state whose current signature specifies the attributes {sname, course}. Now consider the update operation where

$$S_b = \{ (\text{"Phil"}, \text{"English"}), \quad \text{and} \quad S_u = \{ (\text{"Phil"}, \text{"English"}), \\ (\text{"Norman"}, \text{"English"}), \quad (\text{"Norman"}, \text{"English"}), \\ (\text{"Norman"}, \text{"Math"}) \} \quad (\text{"Marilyn"}, \text{"Math"}) \} .$$

Then,

$$\Delta_S = \{ ((\text{"Norman"}, \text{"Math"}), \text{NIL}), \\ (\text{NIL}, (\text{"Marilyn"}, \text{"Math"})) \} .$$

Here, we change the relation state S_b by deleting the tuple ("Norman", "Math") and inserting the tuple ("Marilyn", "Math"). \square

In defining a snapshot differential, we have followed the method of Kahler and Risnes for condensed logging [Kahler & Risnes 1987]. In previous incremental versions of the snapshot algebra [Blakeley et al. 1986A, Hanson 1987A, Horwitz & Teitelbaum 1986], changes to a snapshot state have been represented by two differentials, a positive differential (i.e., the tuples inserted) and a negative differential (i.e., the tuples deleted). Δ_R is simply an encoding of these two differentials as a single differential. We introduce the notion of a single differential now to make the definition of a snapshot differential analogous to that of a historical differential. As we will see later, denoting changes to a historical state as a single differential simplifies somewhat definition of the incremental historical operators.

We can also define a function S_Update that maps a snapshot relation's state just before an update and its differential for that update onto its state immediately after the update.

$$S_Update : [\text{SNAPSHOT STATE} \times \text{SNAPSHOT DIFFERENTIAL}] \rightarrow \\ \text{SNAPSHOT STATE}$$

```

S.Update( $R_b, \Delta_R$ ) =
  if       $\exists r_b \exists r_a, (r_b, r_a) \in \Delta_R$ 
    then   if       $r_b = \text{NIL}$ 
            then   $S.Update(R_b, \Delta_R - \{(r_b, r_a)\}) \cup \{r_a\}$ 
            else    $S.Update(R_b, \Delta_R - \{(r_b, r_a)\}) - \{r_b\}$ 
    else    $R_b$ 

```

$S.Update$ simply applies the changes denoted by the elements of Δ_R to R_b , one at a time. Because Δ_R denotes the net changes to R_b , the order in which the changes are applied is arbitrary. Also, because Δ_R denotes the changes to R_b that produce R_a , R_b and Δ_R together denote R_a .

EXAMPLE. Suppose we let S_b , S_a , and Δ_S be as defined in the previous example. Then, $S.Update(S_b, \Delta_S) = S_a$ holds. \square

6.3.2 Incremental Snapshot Operators

Unfortunately, the incremental snapshot operators can't be defined in terms of differentials alone. As we will see shortly, the output differential for each operator, except that for the selection operator, depends on an input relation's state just before an update as well as the input relation's differential for the update. Hence, both relation states and differentials are required as inputs to the incremental operators. Furthermore, because the output of one operator must be acceptable as input to another operator, the output of each operator must include, for definitional purposes, its output relation's state just before an update, as well as its output relation's differential for the update. Note, however, that this requirement need not be extended to an implementation of the algebra. If an implementation were to cache, either virtually or physically, the input relations to each operator, only differentials would need to be computed and passed among operators. Hence, while an implementation of the incremental algebra based directly on the following formalization is impractical, the algebra can serve as the basis for efficient maintenance of views when incremental materialization is the preferred view maintenance strategy and intermediate results between successive evaluations of the view definition are cached.

We can now define the incremental snapshot operators σ^i , π^i , \cup^i , $-^i$, and \times^i , corresponding to the snapshot operators σ , π , \cup , $-$, and \times , respectively, such that:

- The snapshot state denoted by the snapshot state and differential produced by an incremental unary snapshot operator is equivalent to the snapshot state produced by the corresponding unary snapshot operator.

$uop^i : [SNAPSHOT\ STATE \times SNAPSHOT\ DIFFERENTIAL] \rightarrow$

[SNAPSHOT STATE × SNAPSHOT DIFFERENTIAL]

$$S_Update(uop^1(R, \Delta_R)) \equiv uop(S_Update(R, \Delta_R))$$

- The snapshot state denoted by the snapshot state and differential produced by an incremental binary snapshot operator analogously is equivalent to the snapshot state produced by the corresponding binary snapshot operator.

$$\begin{aligned} bop^1 : [[SNAPSHOT STATE \times SNAPSHOT DIFFERENTIAL] \times \\ [SNAPSHOT STATE \times SNAPSHOT DIFFERENTIAL]] \rightarrow \\ [SNAPSHOT STATE \times SNAPSHOT DIFFERENTIAL] \end{aligned}$$

$$S_Update(bop^1(Q, \Delta_Q, R, \Delta_R)) \equiv bop(S_Update(Q, \Delta_Q), S_Update(R, \Delta_R))$$

Let R be a snapshot state of m -tuples on the relation signature π with attributes $A = \{I_1, \dots, I_m\}$, and Δ_R be a snapshot differential for R . Also, let F be a boolean function as defined in Section 3.3.4. Then incremental snapshot selection is defined as

$$\begin{aligned} \sigma_F^1(R, \Delta_R) \triangleq & (\sigma_F(R), \\ & \{ (\text{NIL}, r^m) \mid (\text{NIL}, r) \in \Delta_R \wedge F(r) \} \\ & \cup \{ (r^m, \text{NIL}) \mid (r, \text{NIL}) \in \Delta_R \wedge F(r) \}) \end{aligned}$$

The output differential contains only those tuples either inserted into or deleted from R that satisfy the predicate. Note that the output differential depends on the input differential and the predicate only; it does not depend on R . Selection is the only incremental snapshot operator that can be defined independently of its input relation state(s).

Now, assume that we are given a set of identifiers X of cardinality n , where $X \subseteq A$.

$$\begin{aligned} \pi_X^1(R, \Delta_R) \triangleq & (\pi_X(R), \\ & \{ (\text{NIL}, u^n) \mid \exists r, ((\text{NIL}, r) \in \Delta_R \wedge \forall I, I \in X, u(I) = r(I) \\ & \quad \wedge \forall r', r' \in R, \exists I, I \in X \wedge r'(I) \neq u(I)) \} \\ & \cup \{ (u^n, \text{NIL}) \mid \exists r, ((r, \text{NIL}) \in \Delta_R \wedge \forall I, I \in X, u(I) = r(I) \\ & \quad \wedge \forall r', (r' \neq r \wedge ((r' \in R \wedge (r', \text{NIL}) \neq \Delta_R) \vee (\text{NIL}, r') \in \Delta_R)), \\ & \quad \exists I, I \in X \wedge r'(I) \neq u(I))) \} \end{aligned}$$

Note that incremental projection, unlike incremental selection, depends on both its input relation state and its input differential. The definition accounts for the possibility that two or more tuples in R can have identical values for attributes X . A tuple inserted into R causes a tuple to be inserted into the projection of R only if there is no other tuple in R 's

old state that has the same values as the inserted tuple for attributes X . Likewise, a tuple deleted from R causes a tuple to be deleted from the projection of R only if there is no tuple in R 's *new* state that has the same values as the deleted tuple for attributes X .

Let Q also be a snapshot state of m -tuples over the relation signature z with attributes $A = \{I_1, \dots, I_m\}$ and let Δ_Q be a snapshot differential for Q .

$$(Q, \Delta_Q) \cup^i (R, \Delta_R) \triangleq (Q \cup R, \\ \{(NIL, u^m) \mid ((NIL, u) \in \Delta_Q \wedge u \notin R) \vee ((NIL, u) \in \Delta_R \wedge u \notin Q)\} \\ \cup \{(u^m, NIL) \mid ((u, NIL) \in \Delta_Q \wedge (NIL, u) \notin \Delta_R \wedge (u \notin R \vee (u, NIL) \in \Delta_R)) \\ \vee ((u, NIL) \in \Delta_R \wedge (NIL, u) \notin \Delta_Q \wedge (u \notin Q \vee (u, NIL) \in \Delta_Q)))\})$$

Incremental union is a symmetric operator; it treats elements in Δ_Q and Δ_R in an identical fashion. A tuple inserted into Q is inserted into $Q \cup R$ only if it is not in R 's *old* state and a tuple deleted from Q is deleted from $Q \cup R$ only if it is not in R 's *new* state. Changes to R are handled analogously.

$$(Q, \Delta_Q) -^i (R, \Delta_R) \triangleq (Q - R, \\ \{(NIL, u^m) \mid ((NIL, u) \in \Delta_Q \wedge (NIL, u) \notin \Delta_R \wedge (u \notin R \vee (u, NIL) \in \Delta_R)) \\ \vee ((u, NIL) \in \Delta_R \wedge ((NIL, u) \in \Delta_Q \vee (u \in Q \wedge (u, NIL) \notin \Delta_Q)))\} \\ \cup \{(u^m, NIL) \mid ((u, NIL) \in \Delta_Q \wedge u \notin R) \vee ((NIL, u) \in \Delta_R \wedge u \in Q)\})$$

Incremental difference, unlike incremental union, is asymmetric. Insertion of tuples into Q causes insertion of tuples into $Q - R$, whereas insertion of tuples into R causes deletion of tuples from $Q - R$. Also, deletion of tuples from Q causes deletion of tuples from $Q - R$, whereas deletion of tuples from R causes insertion of tuples into $Q - R$. A tuple is inserted into $Q - R$ if (1) it is inserted into Q and it is not in R 's *new* state or (2) it is deleted from R and it is in Q 's *new* state. A tuple is deleted from $Q - R$ if (1) it is deleted from Q and it is not in R 's *old* state or (2) it is inserted into R and it is in Q 's *old* state.

Now, let Q be a snapshot state of m_1 -tuples on the relation signature z_Q with attributes $A_Q = \{I_{Q,1}, \dots, I_{Q,m_1}\}$, R be a snapshot state of m_2 -tuples on the relation signature z_R with attributes $A_R = \{I_{R,1}, \dots, I_{R,m_2}\}$, and Δ_Q and Δ_R be snapshot differentials for Q and R , respectively. Also assume that $A_Q \cap A_R = \emptyset$.

$$\begin{aligned}
(Q, \Delta_Q) \times^I (R, \Delta_R) &\triangleq (Q \times R, \\
&\{(NIL, u^{m_1+m_2}) \mid ((\exists q, (NIL, q) \in \Delta_Q \wedge \forall I, I \in A_Q, u(I) = q(I)) \\
&\quad \wedge (\exists r, ((NIL, r) \in \Delta_R \vee (r \in R \wedge (r, NIL) \notin \Delta_R)) \\
&\quad \wedge \forall I, I \in A_R, u(I) = r(I))) \\
&\quad \vee ((\exists r, (NIL, r) \in \Delta_R \wedge \forall I, I \in A_R, u(I) = r(I)) \\
&\quad \wedge (\exists q, ((NIL, q) \in \Delta_Q \vee (q \in Q \wedge (q, NIL) \notin \Delta_Q)) \\
&\quad \wedge \forall I, I \in A_Q, u(I) = q(I))))\} \\
&\cup \{(u^{m_1+m_2}, NIL) \mid ((\exists q, (q, NIL) \in \Delta_Q \wedge \forall I, I \in A_Q, u(I) = q(I)) \\
&\quad \wedge (\exists r, r \in R \wedge \forall I, I \in A_R, u(I) = r(I))) \\
&\quad \vee ((\exists r, (r, NIL) \in \Delta_R \wedge \forall I, I \in A_R, u(I) = r(I)) \\
&\quad \wedge (\exists q, q \in Q \wedge \forall I, I \in A_Q, u(I) = q(I))))\}
\end{aligned}$$

Incremental cartesian product, like incremental union, is symmetric. A tuple inserted into Q causes a tuple to be inserted into $Q \times R$ for each tuple in R 's new state and a tuple deleted from Q causes a tuple to be deleted from $Q \times R$ for each tuple in R 's old state. Changes to R are handled analogously.

6.4 Incremental Historical Algebra

In this section we define an incremental version of our historical algebra in which each operator is defined as a mapping from either one relation state and its differential or two relation states and their differentials onto a resulting relation state and its corresponding differential.

We first define the function *H-Differential* that computes a historical differential and the function *H-Update* that maps a historical relation's state just before an update and its differential for that update onto its state immediately after the update. Then, we define incremental versions of the historical operators introduced in Chapter 3.

6.4.1 Historical Differential

We define the differential for an update operation on a historical relation, like that for an update operation on a snapshot relation, as the set of ordered pairs that records the *before* and *after* images of all tuples that the update changes. Assume that we are given the historical relation R over the relation signature z with attributes $A = \{I_1, \dots, I_m\}$. If we let R_b be R 's state just before an update and R_a be R 's state immediately after the update,

we can define the differential Δ_R for the update in terms of the function $H\text{-Differential}$ as follows.

$$H\text{-Differential} : [HISTORICAL\ STATE \times HISTORICAL\ STATE] \rightarrow HISTORICAL\ DIFFERENTIAL$$

$$\begin{aligned}\Delta_R \triangleq H\text{-Differential}(R_b, R_a) = \\ \{ (r_b, r_a) \mid & (r_b = \text{NIL} \wedge r_a \in R_a) \\ & \wedge \forall r, r \in R_b, \exists I, I \in A \wedge \text{Value}(r(I)) \neq \text{Value}(r_a(I))) \\ & \vee (r_b \in R_b \wedge r_a = \text{NIL} \\ & \wedge \forall r, r \in R_a, \exists I, I \in A \wedge \text{Value}(r(I)) \neq \text{Value}(r_b(I))) \\ & \vee (r_b \in R_b \wedge r_a \in R_a \wedge r_b \neq r_a \\ & \wedge \forall I, I \in A, \text{Value}(r_b(I)) = \text{Value}(r_a(I))) \}\end{aligned}$$

Δ_R denotes the changes to R_b that produce R_a . As before, insertion of a tuple without a value-equivalent counterpart in R_b is denoted by a pair whose first component is NIL and whose second component is the inserted tuple. Deletion of an entire tuple is denoted by a pair whose first component is the deleted tuple and whose second component is NIL . A change to a tuple in R_b that does not require the tuple's deletion (i.e., a change to the valid-time component, but not the value component, of one or more attributes) is denoted by a pair whose first component is the tuple's image before the change and whose second component is the tuple's image after the change. This third possibility, although not present in the snapshot differential, is needed here to record the before and after images of a changed tuple as a single pair. Note that, if both components of a pair are tuples, then the tuples must be value-equivalent, but not equal. Also, if a tuple appears as a component in one pair, then neither it nor any value-equivalent tuple can appear as a component of any other pair. Hence, each pair in the differential denotes an inserted tuple, a deleted tuple, or the net change to a tuple in R_b . Δ_R , like its snapshot counterpart, denotes the net changes to R_b .

EXAMPLE. Let H denote a historical relation whose current signature specifies the attributes {`sname`, `course`}. Now consider the update operation where

$$H_b = \{ \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}) \rangle, \\ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \\ \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}) \rangle \}$$

and

$$H_a = \{ \langle ("Phil", \{3, 4\}), ("English", \{3, 4\}) \rangle, \\ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \\ \langle ("Marilyn", \{3, 4\}), ("Math", \{3, 4\}) \rangle \}.$$

Then,

$$\Delta_H = \{ \langle \langle ("Phil", \{1, 3, 4\}), ("English", \{1, 3, 4\}) \rangle, \\ \langle ("Phil", \{3, 4\}), ("English", \{3, 4\}) \rangle \rangle, \\ \langle \langle ("Norman", \{5, 6\}), ("Math", \{5, 6\}) \rangle, NIL \rangle, \\ (NIL, \langle ("Marilyn", \{3, 4\}), ("Math", \{3, 4\}) \rangle) \}.$$

Here, we change one tuple, delete one tuple, and insert another tuple. \square

We can also define a function H_Update that maps a historical relation's state just before an update and its differential for that update onto its state immediately after the update.

$$H_Update : [HISTORICAL\ STATE \times HISTORICAL\ DIFFERENTIAL] \rightarrow \\ HISTORICAL\ STATE$$

$$H_Update(R_b, \Delta_R) = \\ \text{if } \exists r_b \exists r_a, (r_b, r_a) \in \Delta_R \\ \text{then if } r_b = NIL \\ \text{then } H_Update(R_b, \Delta_R - \{(r_b, r_a)\}) \cup \{r_a\} \\ \text{else if } r_a = NIL \\ \text{then } H_Update(R_b, \Delta_R - \{(r_b, r_a)\}) - \{r_b\} \\ \text{else } H_Update(R_b, \Delta_R - \{(r_b, r_a)\}) - \{r_b\} \cup \{r_a\} \\ \text{else } R_b$$

H_Update simply applies the changes denoted by the elements of Δ_R to R_b , one at a time. Because each element in Δ_R denotes a tuple insertion, a tuple deletion, or the net change to a tuple in R_b , the order in which the changes are applied is arbitrary. Also, because Δ_R denotes the changes to R_b that produce R_a , Δ_R and R_b together denote R_a .

EXAMPLE. If we let H_b , H_a , and the differential Δ_H be as defined in the previous example, then $H_Update(H_b, \Delta_H) = H_a$ holds. \square

6.4.2 Historical Operators

We can now define the incremental historical operators σ^i , δ^i , π^i , \cup^i , \setminus^i , \hat{x}^i , \hat{A}^i , \widehat{AU}^i , \cap^i , $\hat{\cap}^i$, \bowtie^i , and \div^i such that the incremental operators are consistent, as defined by H_Update , with their non-incremental counterparts.

$$\widehat{uop}^i : [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}] \rightarrow [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}]$$

$$H_Update(\widehat{uop}^i(R, \Delta_R)) \equiv \widehat{uop}(H_Update(R, \Delta_R))$$

$$\widehat{bop}^i : [[\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}] \times [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}]] \rightarrow [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}]$$

$$H_Update(\widehat{bop}^i(Q, \Delta_Q, R, \Delta_R)) \equiv \widehat{bop}(H_Update(Q, \Delta_Q), H_Update(R, \Delta_R))$$

As with the incremental snapshot operators, incremental historical operators can't be defined in terms of differentials alone. Their output differentials also depend on an input relation's state just before an update as well as the input relation's differential for the update.

Before defining the operators, we introduce two auxiliary functions, *VECounterpart* and *Unchanged*, which are used in defining two or more of the operators. For their definitions, and the definitions to follow, let Q and R be historical states of m -tuples over the relation signature z with attributes $A = \{ I_1, \dots, I_m \}$ and let Δ_Q and Δ_R be historical differentials for Q and R , respectively.

VECounterpart returns the before and after images of a tuple obtained from a relation's state just before an update and its differential for the update. The tuple returned is the value-equivalent counterpart of a given tuple, where the given tuple is itself specified by its before and after images for an update. If the relation state and differential contain no value-equivalent tuple for the given tuple, *VECounterpart* returns (NIL, NIL).

VECounterpart :

$$[[[\text{HISTORICAL TUPLE} + \{ \text{NIL} \}] \times [\text{HISTORICAL TUPLE} + \{ \text{NIL} \}]] \times [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL}]] \rightarrow [[\text{HISTORICAL TUPLE} + \{ \text{NIL} \}] \times [\text{HISTORICAL TUPLE} + \{ \text{NIL} \}]]$$

$VECounterpart((q_b, q_a), R, \Delta_R) =$

if $((q_b \neq \text{NIL} \wedge q = q_b) \vee (q_b = \text{NIL} \wedge q_a \neq \text{NIL} \wedge q = q_a))$

then if $\exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R$

$\wedge r_b \neq \text{NIL} \rightarrow \forall I, I \in A, \text{Value}(r_b(I)) = \text{Value}(q(I))$

$\wedge r_a \neq \text{NIL} \rightarrow \forall I, I \in A, \text{Value}(r_a(I)) = \text{Value}(q(I)))$

then (r_b, r_a)

else if $\exists r, r \in R \wedge \forall I, I \in A, \text{Value}(r(I)) = \text{Value}(q(I))$

then (r, r)

else (NIL, NIL)

else (NIL, NIL)

EXAMPLE. Let H_b and Δ_H be as defined in the example on page 145.

$VECounterpart((\text{NIL}, \langle(\text{"Norman"}, \{8,9\}), (\text{"Math"}, \{8,9\})\rangle), H_b, \Delta_H) =$

$(\langle(\text{"Norman"}, \{5,6\}), (\text{"Math"}, \{5,6\})\rangle, \text{NIL})$

$VECounterpart((\langle(\text{"Norman"}, \{8,9\}), (\text{"English"}, \{8,9\})\rangle, \text{NIL}), H_b, \Delta_H) =$

$(\langle(\text{"Norman"}, \{1,2\}), (\text{"English"}, \{1,2\})\rangle, \langle(\text{"Norman"}, \{1,2\}), (\text{"English"}, \{1,2\})\rangle)$

$VECounterpart((\langle(\text{"Norman"}, \{8,9\}), (\text{"History"}, \{8,9\})\rangle, \text{NIL}), H_b, \Delta_H) = (\text{NIL}, \text{NIL})$

The first tuple $(\langle(\text{"Norman"}, \{8,9\}), (\text{"Math"}, \{8,9\})\rangle)$ has a value-equivalent counterpart in Δ_H , the second tuple has a value-equivalent counterpart in H_b but not in Δ_H , and the third tuple has a value-equivalent counterpart in neither H_b nor Δ_H . \square

Unchanged determines whether an update operation, as defined by a historical differential, leaves a specified tuple in a historical state unchanged.

Unchanged :

$[\text{HISTORICAL TUPLE} \times \text{HISTORICAL DIFFERENTIAL}] \rightarrow$

$\{\text{TRUE}, \text{FALSE}\}$

Unchanged $(r, \Delta_R) =$

$\forall r_b \forall r_a, (r_b, r_a) \in \Delta_R \wedge r_b \neq \text{NIL} \exists I, I \in A \wedge \text{Value}(r(I)) \neq \text{Value}(r_b(I))$

EXAMPLE. Again, let H_b and Δ_H be as defined in the example on page 145.

$$\text{Unchanged}((\text{"Phil"}, \{1, 3, 4\}), (\text{"English"}, \{1, 3, 4\})) , \Delta_H = \text{FALSE}$$

$$\text{Unchanged}((\text{"Norman"}, \{1, 2\}), (\text{"English"}, \{1, 2\})) , \Delta_H = \text{TRUE}$$

The first tuple is changed by Δ_H , but the second tuple is left unchanged. \square

Given these two auxiliary functions, we can now define an incremental version of each historical operator. If we let F be a boolean function as defined in Section 3.3.4, then incremental historical selection is defined as

$$\delta_F^I(R, \Delta_R) \triangleq (\delta_F(R),$$

$$\{(r_b, r_a) \mid (r_b, r_a) \in \Delta_R \wedge r_b \neq \text{NIL} \rightarrow F(r_b) \wedge r_a \neq \text{NIL} \rightarrow F(r_a)\})$$

Note that the output differential of incremental historical selection, like that of incremental snapshot selection, depends only on the input differential and the selection predicate.

Now, let V_a , $1 \leq a \leq m$, be a temporal function and G be a boolean function as defined in Section 3.3.5.

$$\delta_G^I, \{(I_1, V_1), \dots, (I_m, V_m)\}(R, \Delta_R) \triangleq (\delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\})(R),$$

$$\{(u_b, u_a) \mid \exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R \wedge$$

$$\wedge (r_b = \text{NIL} \vee \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_b\}) = \emptyset) \rightarrow u_b = \text{NIL}$$

$$\wedge (r_b \neq \text{NIL} \wedge \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_b\}) \neq \emptyset) \rightarrow$$

$$u_b \in \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_b\})$$

$$\wedge (r_a = \text{NIL} \vee \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_a\}) = \emptyset) \rightarrow u_a = \text{NIL}$$

$$\wedge (r_a \neq \text{NIL} \wedge \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_a\}) \neq \emptyset) \rightarrow$$

$$u_a \in \delta_G, \{(I_1, V_1), \dots, (I_m, V_m)\}(\{r_a\})$$

$$\wedge u_b \neq u_a))\}$$

The output differential of incremental historical derivation, like that of incremental historical selection, does not depend on the input state R . It depends only on the input differential, the temporal functions V_a , $1 \leq a \leq m$, and the boolean function G . Note that the incremental version of the operator is defined in terms of the non-incremental version of the operator, applied to a subset (here a single tuple) of the original relation state. This approach will be followed in defining the other incremental operators.

Assume that we are given a set of identifiers X of cardinality n , where $X \subseteq A$.

$$\begin{aligned}\pi_X^t(R, \Delta_R) &\triangleq (\pi_X(R), \\ &\{(u_b, u_a) \mid \exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R \\ &\quad \wedge ((r_b \neq \text{NIL} \wedge r = r_b) \vee (r_b = \text{NIL} \wedge r = r_a)) \\ &\quad \wedge u_b = \text{BeforeImage}(R, X, r) \\ &\quad \wedge u_a = \text{AfterImage}(R, \Delta_R, X, r) \\ &\quad \wedge u_b \neq u_a)\})\end{aligned}$$

where *BeforeImage* computes the projected image before update, and *AfterImage* computes the projected image after update, of tuples in R that are value-equivalent to a tuple r for attributes X .

BeforeImage :

$$\begin{aligned}[\text{HISTORICAL STATE} \times [\text{IDENTIFIER}]^* \times \text{HISTORICAL TUPLE}] -> \\ [\text{HISTORICAL TUPLE} + \{\text{NIL}\}]\end{aligned}$$

BeforeImage(R, X, r) =

$$\begin{aligned}\text{if } \exists u, u \in \pi_X(\{r' \mid r' \in R \wedge \forall I, I \in X, \text{Value}(r'(I)) = \text{Value}(r(I))\}) \\ \text{then } u \\ \text{else NIL}\end{aligned}$$

AfterImage :

$$\begin{aligned}[\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL} \times \\ [\text{IDENTIFIER}]^* \times \text{HISTORICAL TUPLE}] -> \\ [\text{HISTORICAL TUPLE} + \{\text{NIL}\}]\end{aligned}$$

AfterImage(R , Δ_R , X , r) =

if $\exists u, u \in \pi_X(\{r' \mid (\exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R \wedge r_a \neq \text{NIL} \wedge r' = r_a \wedge \forall I, I \in X, \text{Value}(r'(I)) = \text{Value}(r(I)))) \wedge (r' \in R \wedge \text{Unchanged}(r', \Delta_R) \wedge \forall I, I \in X, \text{Value}(r'(I)) = \text{Value}(r(I))))\})$

then u

else NIL

Note that incremental historical projection, like incremental snapshot projection, must account for the possibility that two or more tuples in R can have identical value components for attributes X . Hence, the output differential of incremental historical projection, unlike those of incremental historical selection and incremental historical derivation, depends on the input state R , as well as the input differential Δ_R .

$$(Q, \Delta_Q) \dot{\cup}^I (R, \Delta_R) \triangleq (Q \dot{\cup} R,$$

$$\{(u_b, u_a) \mid (\exists q_b \exists q_a, ((q_b, q_a) \in \Delta_Q \wedge (r_b, r_a) = \text{VECounterpart}((q_b, q_a), R, \Delta_R)) \wedge \exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R \wedge (q_b, q_a) = \text{VECounterpart}((r_b, r_a), Q, \Delta_Q))) \wedge (u_b, u_a) = (HUnion(q_b, r_b), HUnion(q_a, r_a)) \wedge u_b \neq u_a\})$$

where $HUnion$ computes the historical union of either the before images or the after images of value-equivalent tuples in Q and R .

$HUnion$:

$$[(\text{HISTORICAL TUPLE} + \{\text{NIL}\}) \times (\text{HISTORICAL TUPLE} + \{\text{NIL}\})] \rightarrow [\text{HISTORICAL TUPLE} + \{\text{NIL}\}]$$

```

HUnion( $u_1, u_2$ ) =
  if  $u_1 \neq \text{NIL}$ 
    then if  $u_2 \neq \text{NIL} \wedge v \in \{u_1\} \dot{-} \{u_2\}$ 
      then  $v$ 
      else  $u_1$ 
    else  $u_2$ 

```

$$\begin{aligned}
(Q, \Delta_Q) \dot{-}^1 (R, \Delta_R) &\triangleq (Q \dot{\cup} R, \\
&\{ (u_b, u_a) \mid (\exists q_b \exists q_a, ((q_b, q_a) \in \Delta_Q \\
&\quad \wedge (r_b, r_a) = \text{VECounterpart}((q_b, q_a), R, \Delta_R)) \\
&\quad \vee \exists r_b \exists r_a, ((r_b, r_a) \in \Delta_R \\
&\quad \wedge (q_b, q_a) = \text{VECounterpart}((r_b, r_a), Q, \Delta_Q))) \\
&\quad \wedge (u_b, u_a) = (\text{HDifference}(q_b, r_b), \text{HDifference}(q_a, r_a)) \wedge u_b \neq u_a \})
\end{aligned}$$

where *HDifference* computes the historical difference of either the before images or the after images of value-equivalent tuples in Q and R .

HDifference :

$$\begin{aligned}
[[\text{HISTORICAL TUPLE} + \{\text{NIL}\}]] \times [[\text{HISTORICAL TUPLE} + \{\text{NIL}\}]] &\rightarrow \\
&[\text{HISTORICAL TUPLE} + \{\text{NIL}\}]
\end{aligned}$$

```

HDifference( $u_1, u_2$ ) =
  if  $u_2 \neq \text{NIL}$ 
    then if  $u_1 \neq \text{NIL} \wedge \{u_1\} \dot{-} \{u_2\} \neq \emptyset \wedge v \in \{u_1\} \dot{-} \{u_2\}$ 
      then  $v$ 
      else  $\text{NIL}$ 
    else  $u_1$ 

```

Now let Q be a historical state of m_1 -tuples on the relation signature α_Q with attributes $A_Q = \{I_{Q,1}, \dots, I_{Q,m_1}\}$, R be a historical state of m_2 -tuples on the relation signature α_R with attributes $A_R = \{I_{R,1}, \dots, I_{R,m_2}\}$, and Δ_Q and Δ_R be historical differentials for Q and R , respectively. Also assume that $A_Q \cap A_R = \emptyset$.

$$(Q, \Delta_Q) \hat{\times}^I (R, \Delta_R) \triangleq (Q \hat{\times} R,$$

$$\{(u_b, u_a) \mid \exists q_b \exists q_a \exists r_b \exists r_a, ((q_b, q_a) \in \Delta_Q \wedge (r_b, r_a) \in \Delta_R)$$

$$\wedge (u_b, u_a) = (HProduct(q_b, r_b), HProduct(q_a, r_a))$$

$$\wedge (u_b, u_a) \neq (\text{NIL}, \text{NIL}))\}$$

$$\cup \{(u_b, u_a) \mid \exists q_b \exists q_a \exists r, ((q_b, q_a) \in \Delta_Q \wedge r \in R \wedge \text{Unchanged}(r, \Delta_R))$$

$$\wedge (u_b, u_a) = (HProduct(q_b, r), HProduct(q_a, r)))$$

$$\vee \exists q \exists r_b \exists r_a, (q \in Q \wedge (r_b, r_a) \in \Delta_R \wedge \text{Unchanged}(q, \Delta_Q))$$

$$\wedge (u_b, u_a) = (HProduct(q, r_b), HProduct(q, r_a))))\}$$

where *HProduct* computes the historical cartesian product of either the before images or the after images of value-equivalent tuples in *Q* and *R*.

HProduct :

$$[[\text{HISTORICAL TUPLE} + \{\text{NIL}\}] \times [\text{HISTORICAL TUPLE} + \{\text{NIL}\}]] \rightarrow [\text{HISTORICAL TUPLE} + \{\text{NIL}\}]$$

$$HProduct(u_1, u_2) =$$

$$\text{if } u_1 \neq \text{NIL} \wedge u_2 \neq \text{NIL} \wedge v \in \{u_1\} \hat{\times} \{u_2\}$$

then *v*

else NIL

Incremental versions of both aggregate operators, \widehat{A}' and \widehat{AU}' , can be defined in terms of \widehat{U}' and $\widehat{\pi}'$. Let *R* be a historical state of *m*-tuples over the relation signature *z* with attributes $A_R = \{I_1, \dots, I_m\}$ and *Q* be a historical state with attributes A_Q , where $A_Q \subseteq A_R$. Also, assume that we are given the scalar aggregate *f*, the windowing function *w*, identifiers *I_a* and *I_agg*, and a set of identifiers *B*, with the restrictions that $I_a \notin B$, $B \cup \{I_a\} \subseteq A_Q$, and $I_{agg} \notin A_Q$. Finally, let *Agg_{t,b}* be the set of tuples input to the projection operator in the definition of \widehat{A} in Section 3.4.2 on page 40 and *Agg_{t,a}* be the set of tuples input to the projection operator in the definition of \widehat{A} if we were to replace all references to *Q* with references to *H_Update(Q, Δ_Q)* and all references to *R* with references to *H_Update(R, Δ_R)*. *Agg_{t,b}* depends on *Q* and *R*, whereas *Agg_{t,a}* depends on *Q, Δ_Q, R*, and Δ_R .

$$\hat{A}_{f, w, I_a, I_{agg}, B}^1(Q, \Delta_Q, R, \Delta_R) \triangleq \\ \bigcup_{t, t \in T(\hat{\pi}_B^1 \cup \{I_{agg}\})}^1 (Agg_{t,b}, H\text{-Differential}(Agg_{t,b}, Agg_{t,a}))$$

Changes to Q have only an isolated affect on the differential for $Agg_{t,b}$. A tuple inserted into, deleted from, or changed in Q may cause an element, representing a change to a tuple in $Agg_{t,b}$, to be included in the differential. A tuple changed in Q , however, causes an element to be included in the differential only if it either satisfied the windowing predicate, as defined by w and t , before the change to Q or satisfies the predicate after the change to Q . A change to R , unlike a change to Q , can have a significant affect on the differential. Whenever a tuple that satisfies the windowing predicate is inserted into, deleted from, or changed in R , new aggregate values must be computed for all tuples in Q that have the same value component as the changed tuple for attributes B . An element will be included in the differential for each tuple in Q whose old and new aggregate values differ. Hence, a change to a tuple in R can cause an arbitrary number of elements to be included in the differential. \widehat{AU}^1 can be defined analogously.

The remaining operators, $\hat{\cap}^1$, $\hat{\otimes}^1$, $\hat{\bowtie}^1$, and $\hat{\div}^1$, all can be defined simply by substituting $\hat{\sigma}^1$, $\hat{\delta}^1$, $\hat{\pi}^1$, \hat{U}^1 , $\hat{\lhd}^1$, and \hat{x}^1 for $\hat{\sigma}$, $\hat{\delta}$, $\hat{\pi}$, \hat{U} , $\hat{\lhd}$, and \hat{x} , respectively, in the definitions of their non-incremental counterparts.

6.5 Language Extensions

Having defined incremental versions of the snapshot and historical algebras, we now extend the language defined in Chapter 4 to accommodate views. We present, in this section, the changes to the language's syntax, semantics domains, and semantic functions **T**, **E**, and **C** that are needed to support views.

6.5.1 Syntax

We need add only three new commands to the language's syntax to accommodate views.

```
C ::= define_view(I, E) | define_recomputed_view(I, E)
      | define_incremental_view(I, E)
```

The command `define_view` creates an unmaterialized view and, as we will see in Section 6.5.5, supports either query modification or in-line view evaluation. The commands `define_recomputed_view` and `define_incremental_view` create materialized views and specify immediate-recomputed and immediate-incremental maintenance strategies, respectively. As stated earlier, we postpone discussion of deferred view materialization until the

next chapter. For all three commands *I* is the identifier that names the view and *E* is the view definition.

6.5.2 Semantic Domains

We need change only one semantic domain to accommodate views. The semantic domain *RELATION* (given on page 61) must be extended to contain a record of (a) whether the database state currently maps an identifier onto a base relation or a view; (b) if a view, the view's definition and whether the view is unmaterialized or materialized; and, (c) if materialized, its maintenance strategy.

$$\begin{aligned}
 \text{RELATION} = & [\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times \\
 & [\text{TRANSACTION NUMBER} + \{-\}]]^* \times \\
 & [\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^* \times \\
 & [[\text{SNAPSHOT STATE} \times \text{TRANSACTION NUMBER}] + \\
 & [\text{HISTORICAL STATE} \times \text{TRANSACTION NUMBER}]]^* \times \\
 & [[\text{EXPRESSION} \times \\
 & \{\text{UNMATERIALIZED, RECOMPUTED, INCREMENTAL}\}] + \{\text{BASE}\}]
 \end{aligned}$$

Note that a relation is now defined as a quadruple, where the fourth component records the needed view-related information. Note also, that a relation's class sequence can now be empty. The relation's class sequence may be empty if the relation is an unmaterialized view because the class of an unmaterialized view is not stored in the database; only the view's definition is stored in the database. A relation's class sequence, however, will be empty only if the relation is an unmaterialized view with no history as either a rollback or a temporal base relation. Also, the class sequence of a materialized view, like that of a base relation, can't be empty because the class of a materialized view, like that of a base relation, is stored in the database.

Because of this extension, all auxiliary functions in Appendix B, defined in terms of the semantic domain *RELATION*, must be extended to handle relations that are quadruples rather than triples. The required changes are minimal and do not change the purpose of the functions.

6.5.3 Type System

The type system for expressions, which was defined in Section 4.2.3, requires only one minor change to accommodate views. Because identifiers can now denote either a base relation or a view, the definition of the semantic function *T* for identifiers (given on page 67) must be extended to handle views as well as base relations.

```

T[I](d, tn) = if      d(I) = (u1, u2, u3, (E, UNMATERIALIZED))
                then   T[E](d, tn)
                else if (LastClass(d(I)) = SNAPSHOT
                           ∨ LastClass(d(I)) = ROLLBACK)
                    then  (SNAPSHOT, LastSignature(d(I)))
                else if (LastClass(d(I)) = HISTORICAL
                           ∨ LastClass(d(I)) = TEMPORAL)
                    then  (HISTORICAL, LastSignature(d(I)))
                else   TYPEERROR

```

If an identifier denotes an unmaterialized view, T determines its type by computing the type of its view definition, since the view's class and signature are not materialized. If, however, the identifier denotes a materialized view, T determines its type just as it would a base relation, since the view's class and signature are materialized.

6.5.4 Expressions

The semantic function E, like the semantic function T, requires one minor change to accommodate views. Its definition for identifiers (given on page 73) must be extended to handle views as well as base relations.

```

E[I](d, tn) = if      (d(I) = (u1, u2, u3, (E, UNMATERIALIZED))
                  ∧ T[I](d, tn) ≠ TYPEERROR)
                then   E[E](d, tn)
                else if T[I](d, tn) ≠ TYPEERROR
                    then  LastState(d(I))
                else   ERROR

```

If an identifier denotes an unmaterialized view, E determines its state by evaluating its view definition, since the view's state is not materialized. If, however, the identifier denotes a materialized view, E determines its state just as it would a base relation, since the view's state is materialized. Note that this change is needed to support in-line evaluation of unmaterialized views. The change would not have been needed if we had required that unmaterialized views be accessed only via query modification. Under query modification, no expression containing a reference to an unmaterialized view is ever evaluated because such expressions are converted to equivalent expressions involving only base relations before

being evaluated.

EXAMPLE. Consider the expression

$$\sigma_{course = "English"}(SP)$$

given in the example on page 134. If we were to evaluate the expression using in-line view evaluation, we would have to evaluate the view definition for the unmaterialized view SP. If, however, we were to convert the expression to the equivalent expression

$$\sigma_{sname = "Phil" \text{ and } course = "English"}(S)$$

using query modification and then evaluate this equivalent expression, we would only have to access the base relation S. \square

Also note that we do not have to extend the definition of E for the rollback operators to account for the presence of views because the definitions given on pages 69 and 71 are sufficient to prevent rollback of a relation to a time when it was a view. A relation can only be rolled back to a time when its class was either rollback or temporal, but a relation's class, when that relation is a view, can only be either snapshot or historical. This restriction is appropriate because views are functions on the current database state.

We do, however, need to introduce a variant of E, which we refer to as E' , to support update of incrementally maintained materialized views. When a database update operation makes changes to a base relation, those changes must be propagated to each materialized view in the base relation's view dependency graph. We use E' in propagating changes to a base relation through that relation's view dependency graph. E' determines, for an incrementally maintained materialized view, the changes that must be made to the view for it to be consistent with its underlying relations when one, or more, of those relations changes.

EXAMPLE. Assume that the views SP, SM, and SU in the example on page 132 were defined as materialized views maintained incrementally, rather than unmaterialized views (i.e., they were defined using the `define_incremental_view` command rather than the `define_view` command). Now consider an update to the base relation S. The changes to S that result from the update must be propagated first to SP and SM and then to SU. The changes that must be made to SP and to SM depend only on the changes to S while the changes that must be made to SU depend on the changes to both SP and SM. To propagate the changes to S through the view dependency graph shown in Figure 6.1, we use E' to determine, given the changes that were made to S, the changes that must be made to SP and SM. Then, we update SP and SM to be consistent with S. Next, we use E' to determine, given the changes that were made to SP and to SM, the changes that must be made to SU. Finally, we update SU to be consistent with SP and SM. In so doing, the changes to S are propagated correctly to SP, SM, and SU. \square

$$\begin{aligned}
 E^I : EXPRESSION &\rightarrow [DATABASE STATE \times DATABASE STATE \times \\
 &\quad TRANSACTION NUMBER] - \\
 &[[SNAPSHOT STATE \times SNAPSHOT DIFFERENTIAL] + \\
 &\quad [HISTORICAL STATE \times \\
 &\quad HISTORICAL DIFFERENTIAL] + \{ ERROR \}]]
 \end{aligned}$$

Unlike E , which maps a semantically correct expression onto a relation state, E^I maps a semantically correct expression onto a relation state and a differential. The relation state is the state, just before a database update operation, of the named or unnamed relation that the expression defines, and the differential is the set of changes that must be made to the relation state to produce the state of the same named or unnamed relation immediately after the update operation. The environment for expression evaluation is the database state just before the update, the database state immediately after the propagation of the changes that result from the update to all relations that the expression references, and the transaction number of the update.

EXAMPLE. Assume that d_b is the state of the database containing S and the incrementally maintained materialized views SP , SM , and SU just before an update to S . Let d_a be the state of the database after the changes that result from the update of S have been propagated to SP and SM . SU 's differential for the update can then be determined by applying E^I to SU 's view definition in the environment defined by d_b , d_a , and the transaction number for the update (i.e., $E^I[\pi(sname)(SP \cup SM)](d_b, d_a, tn)$). \square

We can define the semantic function E^I for each expression in the language as follows.

```

 $E^I[[snapshot, Z, S]](d_b, d_a, tn) =$ 
  if  $T[[snapshot, Z, S]](d_b, tn) \neq \text{TYPEERROR}$ 
  then  $(S[S]Z[Z], \emptyset)$ 
  else  $\text{ERROR}$ 

```

E^I simply maps a snapshot constant onto the snapshot state that it denotes and the empty snapshot differential. E^I for a historical constant is defined analogously.

```

Ei[I](db, da, tn) =
  if      (T[I](db, tn) = T[I](da, tn) = (SNAPSHOT, z))
  then   (E[I](db, tn), S_Differential(E[I](db, tn), E[I](da, tn)))
  else if (T[I](db, tn) = T[I](da, tn) = (HISTORICAL, z))
  then   (E[I](db, tn), H_Differential(E[I](db, tn), E[I](da, tn)))
  else   ERROR

```

An identifier evaluates to the state of the relation that it denotes just before the update and the relation's differential, if any, for the update. Note that the relation must have the same type, as defined by its class and signature, in d_b and d_a . Type-checking is performed because incremental expression evaluation does not allow changes to the type of any relation referenced in an expression; otherwise, a differential could not be computed.

```

Ei[\rho(I, N)](db, da, tn) = if      T[\rho(I, N)](db, tn) ≠ TYPEERROR
                                         then (FindState(db(I), N[N]), ∅)
                                         else ERROR

```

Because updates can't change past states of relations, E^i simply maps an expression involving a snapshot rollback operator onto the state of the relation denoted by I at transaction N and the empty snapshot differential. E^i for historical rollback is defined analogously.

```

Ei[E1 ∪ E2](db, da, tn) =
  if      (T[E1 ∪ E2](db, tn) = T[E1 ∪ E2](da, tn) ≠ TYPEERROR)
  then   Ei[E1](db, da, tn) ∪i Ei[E2](db, da, tn)
  else   ERROR

```

The definition of E^i for snapshot union is formed from the definition of E for snapshot union simply by substituting E^i and its environment for E and its environment and by substituting incremental snapshot union for non-incremental snapshot union. Again, the type of the expression must be the same in both database states. All other snapshot and historical operators are defined analogously.

6.5.5 Commands

We now show the changes to the semantic function C that are needed to accommodate views. We first define C for the three new commands and then extend the definitions of C

for the commands introduced in Section 4.2.5 to take into account the presence of views in the database. Before doing so, however, we describe informally several functions used in the definitions. Formal definitions for these functions appear in Appendix B.

R is a semantic function that maps an expression onto the set of identifiers that occur in the expression.

$BaseRelation$ is a boolean function that determines whether an identifier denotes a defined base relation in a database state.

$MaintenanceStrategy$ maps an identifier that denotes a view in a database state onto the maintenance strategy (i.e., unmaterialized, recomputed, or incremental) for the view. If the identifier does not denote a view, $MaintenanceStrategy$ returns **ERROR**.

$UpdateState$ maps a relation state, differential, and relation class onto the relation state that the input relation state and differential denote. If the class is other than snapshot or historical, $UpdateState$ returns **ERROR**.

$View$ is a boolean function that determines whether an identifier denotes a view, either unmaterialized or materialized, in a database state.

$ViewDef$ maps an identifier that denotes a view in a database state onto the expression that defines the view. If the identifier does not denote a view, $ViewDef$ returns **ERROR**.

$Views$ maps an identifier onto the set of identifiers denoting views that depend, either directly or indirectly, on the relation denoted by the identifier in a database state.

Given these auxiliary functions, we can now define the semantic function C for the new commands.

```

C[define_view(I, E)](d, tn) =
    if      ( $M = MSoT(d(I), tn) \wedge LastClass(d(I)) = \text{UNDEFINED}$ 
             $\wedge T[E](d, tn) \neq \text{ERROR})$ 
        then  ( $d[(M, (E, \text{UNMATERIALIZED}))/I], \text{OK}$ )
        else   (d, ERROR)
    
```

The command `define_view` makes a relation, whose current class is **UNDEFINED**, an unmaterialized view effective when the transaction in which the command occurs is committed. The command simply replaces the relation's first three components with the relation's $MSoT$ and sets the relation's fourth component to the view definition E and the keyword **UNMATERIALIZED**. Neither the view's class, signature, nor state is stored in the database. When the view is accessed, its class, signature, and state will be determined by the semantic functions T and E using E . Note that we only require the view definition to be type-correct. Hence, unmaterialized views can be defined in terms of base relations, materialized views, and other unmaterialized views. Also note that type-checking is sufficient to ensure that the view definition is acyclic. A reference to I in E , either direct or indirect, would produce a type error, thereby aborting the view definition. Finally, storing the view

definition in the database is sufficient to support both query modification and in-line view evaluation. We provide the information needed for query modification (i.e., whether an identifier denotes an unmaterialized view, and if so, its definition) but assume that the actions of query modification are part of the DBMS's user interface, and therefore outside the algebra.

```

C[define_recomputed_view(I, E)](d, tn) =
    if      (M = MSOT(d(I), tn) ∧ LastClass(d(I)) = UNDEFINED
            ∧ T[E](d, tn) = (y, z))
    then   (d[(M ||3((y, tn, -)), NewSignature(M, (z, tn)),
              NewState(M, (E[E](d, tn), tn), (y, z))),
              (E, RECOMPUTED))/I], OK)
    else   (d, ERROR)

```

The command `define_recomputed_view` makes a relation, whose current class is `UNDEFINED`, a materialized view effective when the transaction in which the command occurs is committed. The command also specifies that the view is to be maintained using the immediate-recomputed strategy. Unlike `define_view`, `define_recomputed_view` appends elements to the relation's class, signature, and state sequences, if necessary, to record the view's current class, signature, and state values. Hence, for retrieval, the view can be treated the same as a base relation. Materialized views, like unmaterialized views, can be defined in terms of base relations, unmaterialized views, and other materialized views, although practical considerations make definition of a materialized view in terms of an unmaterialized view improbable. Note that a materialized view's class, as well as that of an unmaterialized view, is either snapshot or historical. A view's class can't be rollback or temporal, as views are functions on the current database state. A relation currently defined as a view, however, can have a (transaction-time) history as a rollback or temporal base relation, accessible via rollback.

```

C[define_incremental_view(I, E)](d, tn) =
    if      (M = MSOT(d(I), tn) ∧ LastClass(d(I)) = UNDEFINED
            ∧ T[E](d, tn) = (y, z))
    then   (d[(M ||3((y, tn, -)), NewSignature(M, (z, tn)),
              NewState(M, (E[E](d, tn), tn), (y, z))),
              (E, INCREMENTAL))/I], OK)
    else   (d, ERROR)

```

The command `define_incremental_view` is identical to `define_recomputed_view`, with

one exception: the view is to be maintained using the immediate-incremental strategy rather than the immediate-recomputed strategy. Note, however, that computation of the view's initial state value is non-incremental.

We now can redefine the four commands, originally defined in Section 4.2.5, to accommodate views.

```
C[define_relation(I, Y, Z)](d, tn) =
  if      (M = MSoT(d(I), tn) ∧ LastClass(d(I)) = UNDEFINED
           ∧ Y[Y] ≠ ERROR ∧ Z[Z] ≠ ERROR)
    then if   FindClass ((M, BASE), tn - 1) = Y[Y]
      then (d[(Expand(M) ||3 (( ), NewSignature(M, (Z[Z], tn)),
                                         NewState(M, (Ø, tn), (Y[Y], Z[Z]))),
                                         BASE)/I], OK)
            else (d[(M ||3 ((Y[Y], tn, -)), NewSignature(M, (Z[Z], tn)),
                                         NewState(M, (Ø, tn), (Y[Y], Z[Z]))),
                                         BASE)/I], OK)
    else (d, ERROR)
```

Only minor changes are needed to the `define_relation` command (c.f., page 80) to accommodate views. The keyword `BASE` is added as the fourth component of the relation to record that a base relation is being defined. Also the relation's `MSoT` is augmented with this same keyword before being passed as an argument to the function `FindClass`.

The `modify_relation` command (c.f., page 83), however, requires more extensive changes because each change to a base relation now must be propagated to every materialized view in the relation's view dependency graph.

EXAMPLE. Assume that the views `SP`, `SM`, and `SU` in the example on page 132 were defined as incrementally maintained materialized views. Then, if `S` were updated to include the tuple `{"Marilyn", "Math"}`, `SM` would have to be updated to include the same tuple, and `SU` would have to be updated to include the tuple `{"Marilyn"}`. □

```

C[modify_relation(I, Y', Z', E)](d, tn) =
  if      (M = MSoT(d(I), tn) ∧ T[E](d, tn) ≠ ERROR ∧ BaseRelation(I, d)
    ∧ Consistent(Y'[Y'](d(I)), Z'[Z'](d(I)), T[E](d, tn))
    ∧ da = UpdateViews(d, d[(M ||3 ((Y'[Y'](d(I), tn, -)), ((Z'[Z'](d(I)), tn),
      ((Σ[E](d, tn), tn))), BASE)/I], tn, Views(I, d)))
  then if    FindClass((M, BASE), tn - 1) = Y'[Y'](d(I))
    then (da[(Expand(M) ||3 (( ), NewSignature(M, (Z'[Z'](d(I), tn)),
      NewState(M, (E[E](d, tn), tn), T[E](d, tn))),,
      BASE)/I], ok)
  else   (da[(M ||3 ((Y'[Y'](d(I), tn, -)), NewSignature(M, (Z'[Z'](d(I), tn)),
      NewState(M, (E[E](d, tn), tn), T[E](d, tn))),,
      BASE)/I], ok)
  else  (d, ERROR)

```

We added two predicates, denoted by the functions *BaseRelation* and *UpdateViews*, to the definition of *modify_relation* to accommodate views. The predicate *BaseRelation* ensures that the relation being changed is a base relation and type-checking of view definitions within *UpdateViews* ensures that all views that depend, either directly or indirectly, on the relation are consistent with the relation's class and signature after the change. Otherwise, the change is not allowed. If all view definitions are consistent with the base relation's type after the change, materialized views that depend on the relation are updated to reflect the change. *UpdateViews* also performs this task. As with *define_relation*, we add the keyword **BASE**, where appropriate, to record that the relation being changed is a base relation.

UpdateViews takes four arguments: (a) the database state just before a base relation is updated, (b) the database state immediately after the relation has been updated and the changes to the relation have been propagated to zero or more of the views in the relation's view dependency graph, (c) the transaction number for the update, and (d) the set of views from the relation's view dependency graph that have yet to be updated. *UpdateViews* updates materialized views in the set to account for changes to their underlying relations that result from the update of the base relation and verifies that the view definitions of unmaterialized views are consistent with the class and signature of each of their underlying relations. If the definition of any view, either unmaterialized or materialized, is inconsistent with the class or signature of one of its underlying relations, *UpdateViews* returns **ERROR**.

Update Views :

$[DATABASE\ STATE \times DATABASE\ STATE \times TRANSACTION\ NUMBER] \times [P(IDENTIFIER)] \rightarrow [DATABASE\ STATE + \{ERROR\}]$

Update Views(d_b, d_a, tn, v) =

if $v \neq \emptyset$

then if $\exists I, (I \in v \wedge E = ViewDef(I, d_a) \wedge v \cap R[E] = \emptyset$

$\wedge M = MSOT(d_a(I), tn) \wedge T[E](d_a, tn) = (y, z)$

$\wedge MaintenanceStrategy(I, d_a) = UNMATERIALIZED \rightarrow d'_a = d_a$

$\wedge (MaintenanceStrategy(I, d_a) = RECOMPUTED$

$\vee (MaintenanceStrategy(I, d_a) = INCREMENTAL$

$\wedge \exists I', (I' \in R[E] \wedge T[I'](d_b, tn) \neq T[I'](d_a, tn))) \rightarrow$

$d'_a = d_a [(M ||_3 (((y, tn, -)), NewSignature(M, (z, tn))),$

$NewState(M, (E[E](d_a, tn), tn), (y, z))),$

$(E, MaintenanceStrategy(I, d_a))/I]$

$\wedge (MaintenanceStrategy(I, d_a) = INCREMENTAL$

$\wedge \forall I', I' \in R[E], T[I'](d_b, tn) = T[I'](d_a, tn)) \rightarrow$

$d'_a = d_a [(M ||_3 (((y, tn, -)), NewSignature(M, (z, tn))),$

$NewState(M, (UpdateState(E'[E](d_b, d_a, tn), y), tn), (y, z))),$

$(E, INCREMENTAL))/I]]$

then *Update Views*($d_b, d'_a, tn, v - \{I\}$)

else **ERROR**

else d_a

Update Views selects from v a view that depends on no other view in v (i.e., $v \cap R[E] = \emptyset$). It then type-checks the view's definition, whether the view is unmaterialized or materialized. This action alone is sufficient to determine whether the view's definition is consistent with the type of each of its underlying relations. No further action is taken if the view is unmaterialized. If the view is materialized, however, it is updated to reflect any changes to

its underlying relations that resulted from the update of the base relation. If the immediate-recomputed strategy is specified or the immediate-incremental strategy is specified, but the type of at least one of the view's underlying relations has changed, the new state of the view is computed using E . Only if the immediate-recomputed strategy is specified and the type of none of the view's underlying relations has changed, which is likely to be the most common situation, is the new state of the view computed using E^t . After type-checking the view's definition and updating the view, if materialized, *Update Views* removes the view from v and calls itself recursively to update the remaining views.

EXAMPLE. Suppose a *modify_command* updates base relation S in the example on page 132. Assuming SP , SM , and SU are incrementally maintained materialized views, *Update Views* would be executed, following the update of S , for the set $\{SP, SM, SU\}$. *Update Views* would select either SP or SM for update, as neither is defined in terms of other views in the set. Say, for the sake of discussion, that SM is selected. *Update Views* would update SM and then call itself recursively for the set $\{SP, SU\}$. During its second execution, *Update Views* would select SP for update and, after updating SP , call itself recursively once more for the set $\{SU\}$. During its third execution, *Update Views* would update SU and return the database state resulting from the propagation of the changes to S to SP , SM , and SU . Note that the order in which the views are selected for update corresponds to a topological sort (i.e., an ordering of the nodes in a directed acyclic graph such that for all edges (u, v) , node u comes before node v in the ordering) of the view dependency graph for the base relation S . \square

```
C[destroy(I)](d, tn) =
  if      ( $M = MSoT(d(I), tn) \wedge (BaseRelation(I, d) \vee View(I, d))$ 
           $\wedge Views(I, d) = \emptyset$ )
  then   ( $d[(M \parallel_3 (((UNDEFINED, tn, -)), ( ), ( )), BASE)/I], \text{OK}$ )
  else   ( $d, \text{ERROR}$ )
```

The **destroy** command (c.f., page 86) is extended to delete either a base relation or a view. Also, one new condition must hold. For a relation, whether it be a base relation or a view, to be deleted, there must be no views that depend on the relation. Otherwise, the deletion is not allowed.

```

C[rename_relation(I1, I2)](d, tn) =
  if      (BaseRelation(I1, d)  $\wedge$  LastClass(d(I2)) = UNDEFINED
     $\wedge$  Y[Y] = LastClass(d(I1))  $\wedge$  Z[Z] = LastSignature(d(I1))
     $\wedge$  C[define_relation(I2, Y, Z)](d, tn) = (d', OK)
     $\wedge$  C[modify_relation(I2, *, *, I1)](d', tn) = (d'', OK)
     $\wedge$  C[destroy(I1)](d'', tn) = (d''', OK))
  then   (d''', OK)
  else if (View(I1, d)  $\wedge$  LastClass(d(I2)) = UNDEFINED  $\wedge$  ViewDef(I1, d) = E
     $\wedge$  ( (ViewStrategy(I1, d) = UNMATERIALIZED
       $\wedge$  C[define_view(I2, E)](d, tn) = (d', OK))
     $\vee$  (ViewStrategy(I1, d) = RECOMPUTED
       $\wedge$  C[define_recomputed_view(I2, E)](d, tn) = (d', OK))
     $\vee$  (ViewStrategy(I1, d) = INCREMENTAL
       $\wedge$  C[define_incremental_view(I2, E)](d, tn) = (d', OK)))
     $\wedge$  C[destroy(I1)](d', tn) = (d'', OK))
  then   (d'', OK)
  else   (d, ERROR)

```

The `rename_relation` command (c.f., page 87) is extended to rename views as well as base relations. Because a relation can't be deleted if there are views that depend on it, a relation, likewise, can't be renamed if there are views that depend on it.

6.6 Scheme Evolution in the Presence of Views

The presence of views restricts the changes that are allowed to the scheme of base relations. In Chapter 4 the `modify_relation` command allowed arbitrary changes to a base relation's class, signature, and state as long as the relation's class, signature, and state remained consistent (i.e., the type of the relation, as defined by its class and signature, was the same as the type of the expression that denoted the relation's state). The presence of views places no additional restrictions on changes to a base relation's state. All changes to a base relation's state that are consistent with the relation's class and signature are still allowed. The presence of views, however, restricts the changes that are allowed to a base relation's class and signature.

Class changes between snapshot and rollback and between historical and temporal

are always allowed because these changes in a relation's class do not result in a change in the relation's type. Class changes between snapshot or rollback and historical or temporal, however, are allowed only if the view definition of no view that depends on the relation contains a snapshot or rollback operator. If there is one view that depends, either directly or indirectly, on the base relation and that view's definition contains either a snapshot or historical operator, these class changes are not allowed as they would cause a type error for the view definition.

Changes to a base relation's signature (i.e., inserting an attribute, deleting an attribute, changing an attribute's value domain, and renaming an attribute) are allowed if the changes do not cause a type error for the view definition of any view that depends on the base relation. Changes to a base relation's signature, when they are allowed, however, cause incrementally maintained views that are defined in terms of the base relation to be updated non-incrementally.

EXAMPLE. Consider the base relation *S* and the views *SP*, *SM*, and *SU* from the example on page 132. Addition of a new attribute to *S*'s signature or deletion of the attribute *course* from *S*'s signature would cause a change to *SP*'s and *SM*'s type but would not cause a type error. Hence, these changes to *S*'s signature would be allowed. But, if the views were being incrementally materialized, their new states would be recomputed rather than incrementally updated for these changes. Deletion of the attribute *sname* from *S*'s signature, however, would cause a type error in the view definition of all three views. Hence, this change would not be allowed. □

The requirement that views be consistent with their underlying relations after the execution of each command rather than after the execution of each transaction, disallows multiple-command transactions that would leave the database in a consistent state after the execution of the transaction but not after the execution of each command in the transaction.

EXAMPLE. Suppose *SP* and *SM* were base relations. Consider a two-command transaction whose first command adds a new attribute to *SP* and whose second command adds the same attribute to *SM*. Although *SU* would be type-correct after the execution of both commands, it would not be type-correct after the execution of the first command because *SP* and *SM* would not be union-compatible. Hence, this transaction would be aborted. □

Type-checking of view definitions within *ViewUpdate* is sufficient to ensure that the restrictions on schema evolution are enforced for all base relations upon which views depend. Also, none of the restrictions are applied to a base relation if there are no views defined in terms of that base relation.

6.7 Summary

In this chapter we have extended the language defined in Chapter 4 to accommodate views. Support for views required changes in the language's syntax and semantics. Three new commands, which define views and specify view maintenance strategies, were added to the language's syntax. Elements of the semantic domain *RELATION* were redefined to

include view-related information and the semantics functions T and E were extended to handle views. The semantic function C was defined for the new commands and redefined for existing commands to account for the presence of views. Also, incremental versions of the snapshot and historical algebras were defined to support incrementally materialized views and a new semantic function E^I was introduced to handle incremental expression evaluation. The incremental snapshot algebra is simply a restatement of the algorithms for incremental update presented elsewhere [Blakeley et al. 1986A, Hanson 1987A, Horwitz & Teitelbaum 1986]. The incremental historical algebra, however, is new. As far as we know, this is the first effort to define an incremental version of a historical algebra. In applying the concepts of incremental expression evaluation to our historical algebra, we found that our algebra is as amenable as the snapshot algebra to incremental expression evaluation.

The contribution of this work is support for views, and a range of view maintenance strategies, in the context of general support for temporal databases. Both unmaterialized and materialized views are supported, as are the view maintenance strategies of query modification, in-line view evaluation, immediate-recomputed materialization, and immediate-incremental materialization. This support for views is achieved without loss of language capability or expressiveness and with only minor changes to the language's syntax and semantics. Although the language still supports arbitrary changes to both the scheme (i.e., class and signature) and state of base relations, the presence of views does restrict the changes to a base relation's scheme that are allowed, but only if there are views that are defined in terms of that base relation.

In the next chapter we discuss an architecture appropriate for the incremental maintenance of views in temporal databases.

Chapter 7

Incremental View Materialization

In the previous chapter we added support for views to our language for query and update of temporal databases. We now describe an architecture for query processing in TDBMS's that accommodates incremental maintenance of materialized historical views. This architecture is an adaptation of an existing architecture for query processing in conventional RDBMS's that accommodates incremental maintenance of snapshot views.

7.1 Background

A *view definition* is simply the algebraic expression that defines the scheme and state of a view. Hence, the problem of materializing a view reduces to that of evaluating an algebraic expression. In the traditional paradigm for expression evaluation, an expression's parse tree is generated and then reduced to a relation state by recursively replacing a subtree rooted at an interior node, whose children are all relation states, with the relation state denoted by the subtree [Maier 1983].

EXAMPLE. Let S_1 denote a snapshot relation whose current signature specifies the attributes {*sname*, *course*} and S_2 denote a snapshot relation whose current signature specifies the attributes {*hname*, *state*}. Now consider the view S_3 defined by the following `define_view` command.

```
define_view(S3, π(sname, state)(σsname=hname(S1×S2)))
```

S_3 's parse tree and the steps in its reduction during expression evaluation are shown in Figure 7.1. T_1 and T_2 are the intermediate results of the evaluation. Also, circles denote relation states while rectangles denote operator nodes. \square

While this paradigm is adequate for implementing recomputed view materialization, it is inadequate for implementing incremental view materialization. The paradigm cannot be used to identify, without recomputing a view itself, the tuples that must be inserted into,

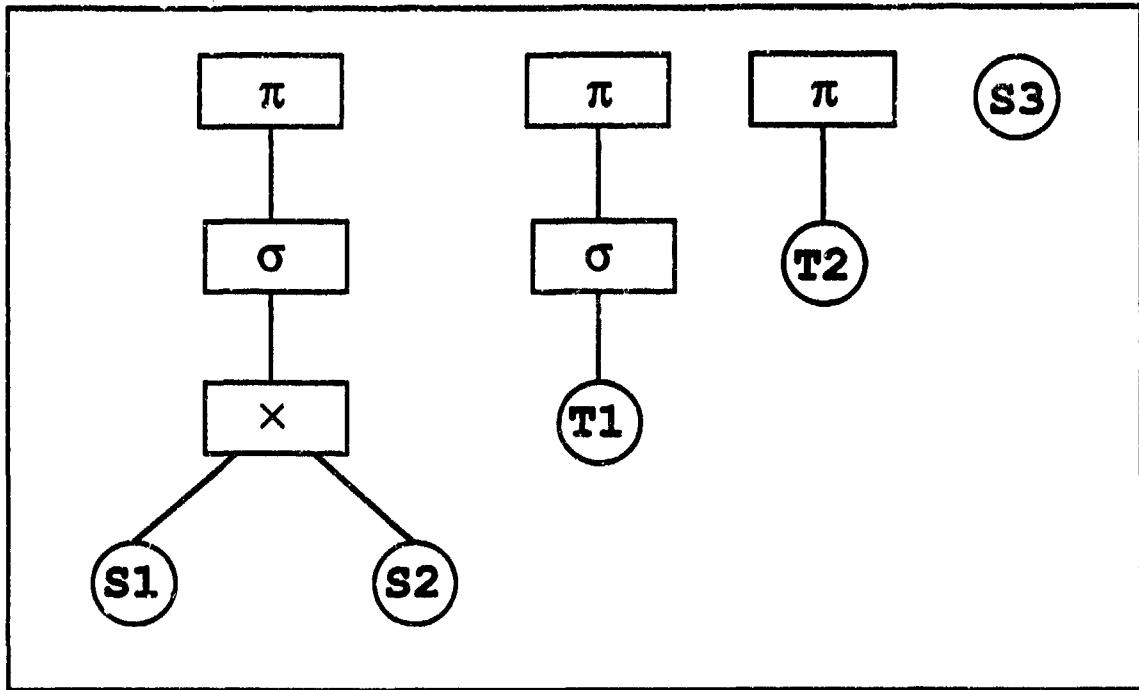


Figure 7.1: Parse Tree for View S3

or the tuples that must be deleted from, the view's old state for the view's new state to be consistent with the new states of its underlying relations following their update.

Snodgrass, Horwitz, and Roussopoulos all have studied the problem of implementing incremental view materialization as a view maintenance strategy [Horwitz 1985, Horwitz & Teitelbaum 1986, Roussopoulos & Kang 1986A, Roussopoulos & Kang 1986B, Roussopoulos 1987, Snodgrass 1982]. In so doing, they independently have proposed variations of a paradigm for incremental expression evaluation. This paradigm uses an expression's parse tree as the basis for building a processing network appropriate for incremental expression evaluation.

Snodgrass and Horwitz both propose that a view definition be mapped onto an acyclic graph of processing nodes, which Snodgrass refers to as the view's *update network*. (We also follow this convention hereafter). The update network has the form of a parse tree, where each node performs the function of an incremental snapshot operator and differentials are passed between nodes via edges. Differentials for the view's underlying relations, when input to the network, cause the corresponding differential for the view to be output from the network.

EXAMPLE. Let S3 be the view from the previous example, declared using the `define_incremental_view` command rather than the `define_view` command. Its update network is shown in Figure 7.2. We have elected to show the network as an inverted parse tree with directed edges to emphasize the flow of differentials through the network. Note that nodes are now labeled with incremental snapshot operators rather than their non-incremental

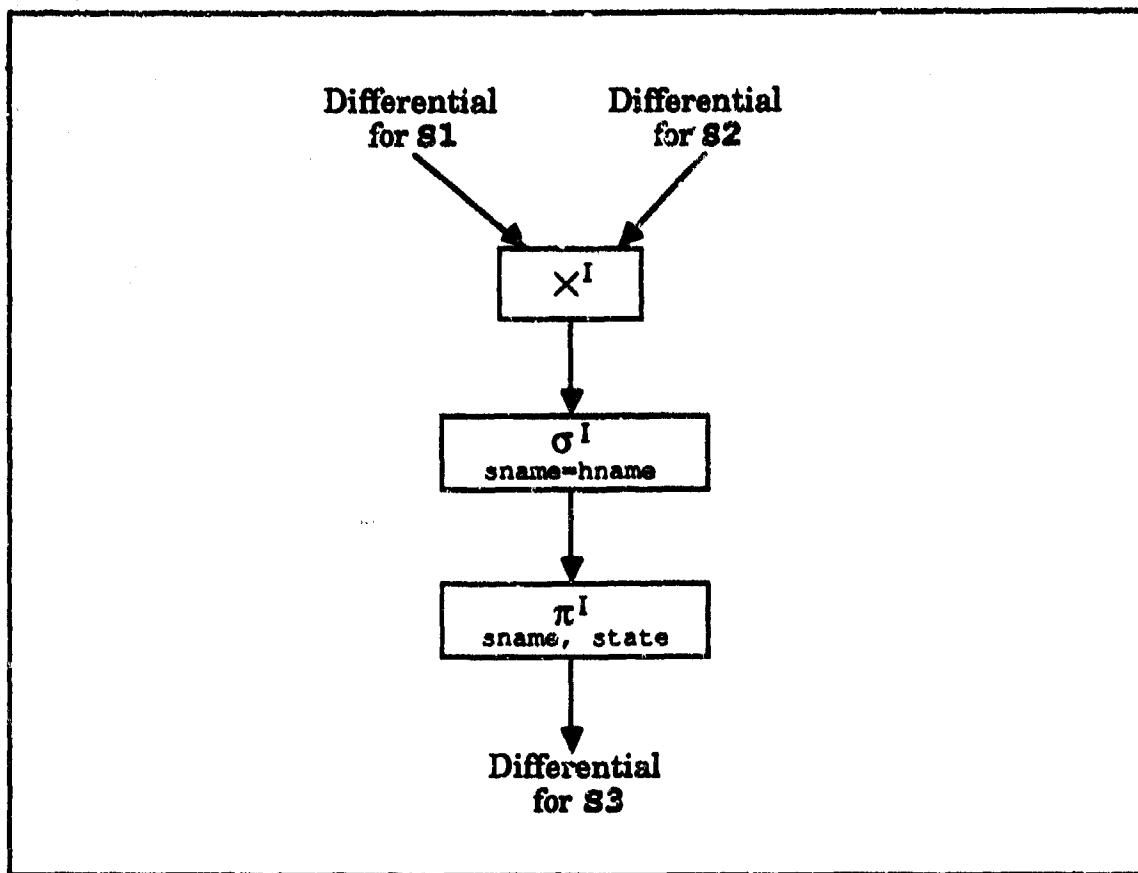


Figure 7.2: Update Network for View S3 As Formalized by Horwitz and Snodgrass

counterparts. Note also that each node can be thought of as computing a relation state incrementally. If we were to apply the differentials output by the cartesian product node to an initially empty relation state, we would materialize the relation state denoted by $S_1 \times S_2$, which would correspond to the relation state T_1 in Figure 7.1. Likewise, if we were to apply all the differentials ever output by the projection node to an initially empty relation state, we would materialize the view itself. \square

This paradigm for incremental expression evaluation differs fundamentally from that for non-incremental expression evaluation. First, the update network, unlike the parse tree, is persistent. It is built when a view is defined, activated each time one of the view's underlying relations is changed, and destroyed only when the view itself is deleted from the database. Second, operator nodes may have their own local memory and procedures. For example, intermediate results from one activation of the network may be cached in operator nodes for use in the next activation of the network. Caching intermediate results at operator nodes between activations of the network is one way to implement all the incremental operators defined in the previous chapter, while passing only differentials between nodes. (Note that "caching" here refers to the storage of intermediate results in a node's local memory, whether that memory is a volatile cache or a stable store.) Third, the input

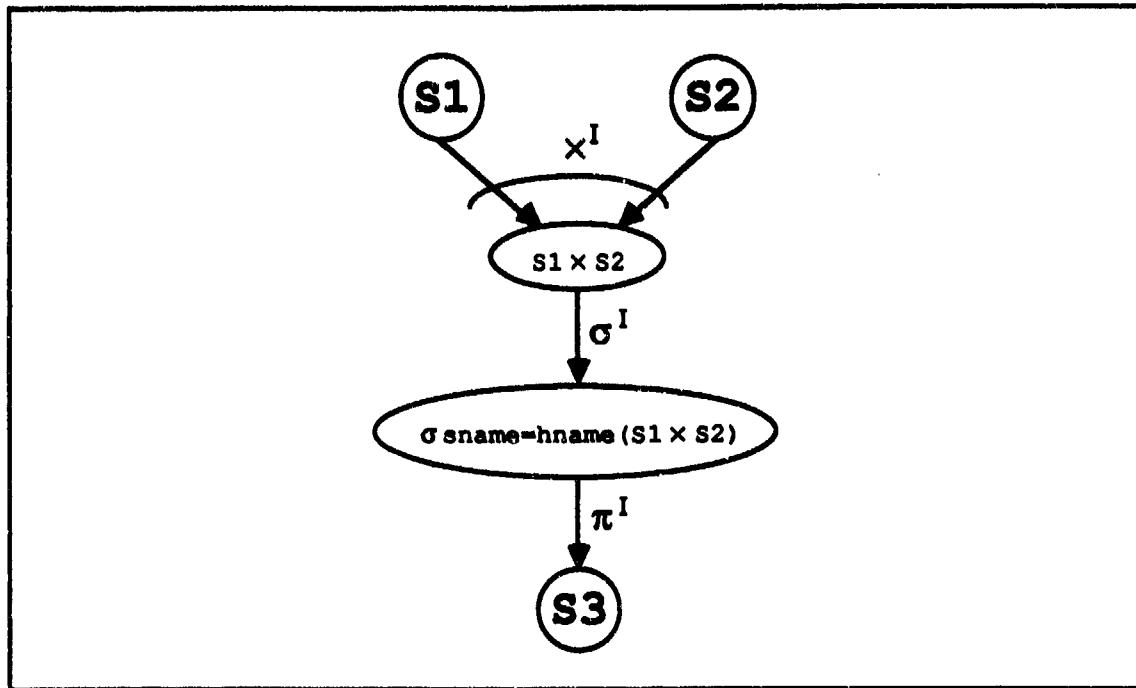


Figure 7.3: Update Network for View S3 As Formalized by Roussopoulos

to, and the output from, the update network is defined in terms of differentials rather than relation states.

Roussopoulos' formalization of the paradigm for incremental expression evaluation differs only slightly from those proposed by Snodgrass and Horwitz. In Roussopoulos' formalization, nodes denote either base relations or views (all intermediate results are treated as views) and edges denote incremental operators. Figure 7.3 shows the update network for S3 as formalized by Roussopoulos.

In addition to proposing a paradigm for incremental expression evaluation, Snodgrass, Horwitz, and Roussopoulos have studied problems that arise when the paradigm is used to implement incremental view materialization. Also, each describes techniques that can be used to improve the performance of update networks in various processing environments.

Snodgrass has studied incremental maintenance of materialized views in the context of monitoring [Snodgrass 1982]. When a computational process is monitored, data are generated by sensors, collected by a resident monitor, and passed to a remote monitor where they eventually become inputs to user-defined queries. Snodgrass argues that monitoring data should be processed as they are collected rather than at the end of a monitoring period, so that data collection and query evaluation can be done in parallel and query results can be presented to the user in (somewhat delayed) real-time [Snodgrass 1987]. To support this capability, he advocates that queries against monitoring data be treated as incrementally maintained materialized views.

Snodgrass shows how to map a TQuel query, when implemented as a view, onto an

update network and describes 12 types of operator nodes useful in processing monitoring data incrementally. Included are operator nodes that perform selection, projection, cartesian product, and join operations incrementally. Because TQuel conceptually embeds a relation's contents in a snapshot relation state (c.f., Chapter 5), only nodes that implement incremental snapshot operators are considered. Snodgrass also discusses techniques that can be used to improve the space and time efficiency of update networks; some are specific to monitoring while others apply equally to other processing environments. These latter techniques include the use of query optimization techniques in building efficient update networks, design of appropriate data structures for each type of node that requires local storage of intermediate results, propagation of differentials using depth-first search, and compilation of update networks.

Horwitz has studied incremental maintenance of materialized views in the context of language-based editing environments [Horwitz 1985, Horwitz & Teitelbaum 1986]. Language-based editing environments are used to detect and prevent programming errors during program entry. Horwitz describes a language-independent model of editing environments based on attribute grammars and the relational data model. In her model, programs are attributed abstract-syntax trees while relations record information needed for the detection and prevention of errors (e.g., static-semantic checking, anomaly detection, and program interrogation), information normally scattered throughout the program tree. Because the relations recording these aggregate data may need to be updated after every editing operation, Horwitz advocates that the relations be implemented as incrementally maintained materialized views.

Horwitz formally defines incremental versions of eight snapshot operators: union, difference, intersection, is-in, equi-join, cartesian product, selection, and projection. She also proposes a technique for implementing all these incremental operators, except cartesian product, as nodes in an update network without having to pass relation states between nodes or having to cache intermediate results at the nodes between activations of the network. To support this implementation strategy, she defines three procedures for each incremental operator: *membership-test*, which determines whether a tuple is in the operator's output relation state; *selective-retrieval*, which returns the tuples in the operator's output relation state that match values specified for some subset of attributes; and *relation-producing*, which builds the operator's output relation state. These procedures can be used by operator nodes to answer tuple membership questions and perform selective tuple retrievals on their input relation states without having to access the relation states themselves. To answer membership questions or perform selective tuple retrievals on an input relation state, a node simply calls the *membership-test* or *selective-retrieval* procedure for the operator node that computes differentials for the input relation state in question.

The primary advantage of this strategy for implementing incremental view materialization is that it avoids cacheing of intermediate results at most operator nodes between network activations. The approach may provide both time and space savings over cacheing of intermediate results, as most nodes in the network would be memoryless. Hence, it is suited to systems, such as in-core database systems [Lehman & Carey 1986], in which

processing time and temporary storage are concerns [Horwitz 1985]. The approach, however, has three disadvantages. First, a call to a procedure at one level of the network causes recursive calls to procedures at each preceding level in the network until a node whose input relation state is cached or a base relation is encountered. Second, cartesian product nodes are still required to cache their input relation states between activations of the network. Third, projection nodes have to call membership-test procedures with wild-card values. The presence of wild-card values as arguments affects the cost of calling the membership-test of several operators, including selection. Horwitz compares the cost of using membership-test and selective-retrieve procedures to the cost of cacheing intermediate results. She concludes that at least the input relation states for cartesian product and projection nodes should be cached to implement incremental view materialization at a reasonable cost.

Roussopoulos has studied incremental maintenance of materialized views in ADMS \pm , an extended centralized architecture for databases which integrates a mainframe database system, called ADMS+, and a workstation database system, called ADMS- [Roussopoulos & Kang 1986A, Roussopoulos & Kang 1986B, Roussopoulos 1987]. ADMS \pm is not a distributed DBMS, but rather a centralized DBMS in which a tailored subset of the database is downloaded to each workstation for local processing. Hence, ADMS \pm provides a centralized database environment, but distributes data and processing to workstations. Base relations and views, once downloaded to a workstation, are maintained using a deferred-incremental update strategy. Changes to base relations and views on the mainframe are recorded in relation backlogs but are not broadcast to workstations. Only when a user at a workstation attempts to access the outdated local copy of a downloaded relation is a differential for that relation transmitted to the workstation and the relation updated.

Views in ADMS \pm are implemented as update networks. Update networks, and portions of update networks, that are common to many workstations reside on the mainframe, while update networks, and portions of update networks, that are common to only a few workstations reside on their workstations. Unlike Horwitz, Roussopoulos advocates that the output relation state of each incremental operator in an update network be cached between activations of the network. Indexing, however, is used to reduce the cost of storing and maintaining the cache [Roussopoulos 1982B, Roussopoulos 1987]. While base relations are materialized, views and intermediate relation states are maintained as indexes. The output relation for an operator, other than cartesian product or join, is stored as a vector, where each element is the address of a tuple in one of the operator's input relation states that contributes to a tuple in its output relation state. The output relation for a cartesian product or join operator is stored as a two-dimensional matrix of address pairs, where the elements of each pair are addresses of tuples in the operator's input relation states that contribute to a tuple in its output relation state. Because an update network is a directed acyclic graph rooted at base relations, addresses always point, either directly or indirectly, to tuples in base relations, the level of indirection determined by the depth of the operator in the update network. Although this strategy is space efficient, retrieval of a tuple in a view or an intermediate relation state requires that tuples in base relations be fetched via one or more levels of indirection and then mapped onto the desired tuple, using the oper-

ators that appear on the path leading from the base relations to the view or intermediate relation state.

7.2 Approach

Our goal in this chapter is to show that the paradigm for incremental expression evaluation independently proposed by Snodgrass, Horwitz, and Roussopoulos can be used, along with the incremental snapshot and historical algebras defined in the previous chapter, to implement incremental view materialization in TDBMS's. The adequacy of the paradigm and incremental snapshot algebra for incrementally maintaining materialized snapshot views has already been shown [Horwitz 1985, Roussopoulos 1987, Snodgrass 1982]. To show the adequacy of the paradigm and historical algebra for incrementally maintaining materialized *historical* views, we built a prototype query processor for TQuel. In this prototype, an update network, defined in terms of incremental historical operators, is used to update materialized views incrementally following changes to their underlying relations. Construction of the prototype is proof that the incremental historical algebra defined in the previous chapter is sufficient to support the incremental evaluation of standard TQuel queries. To be useful, update networks, in addition to being correct, must also be efficient [Snodgrass 1982]. Hence, we will discuss implementation issues that arise when update networks contain nodes that implement incremental historical operators. We describe several techniques that can be used to improve the performance of such networks. In so doing, we examine the applicability of existing optimization techniques, which can be used to improve the performance of update networks containing incremental snapshot operators, to update networks containing incremental historical operators.

We emphasize implementation issues because of the potential importance of this view maintenance strategy to query processing in TDBMS's. Queries in TDBMS's can be grouped into three broad classes: snapshot queries, rollback queries, and non-rollback, historical queries. Snapshot queries involve neither valid time nor transaction time; Ahn has shown that this class of queries can be supported in TDBMS's without performance penalty if appropriate storage structures are used [Ahn 1986A]. Rollback queries, which reference either rollback or temporal relations, are queries asked "as of" some time in the past. Because the past states of rollback and temporal relations never change, both the cost and result of processing a rollback query is constant over time. If a rollback query's execution frequency is sufficiently high, it is cost-effective to evaluate the query once and cache the result for future reference. Otherwise, it is cost-effective to simply re-evaluate the query each time it is asked.

Historical queries are queries on the current state of historical and temporal relations. Because the size of the current state of historical and temporal relations is likely to increase monotonically over time, the cost of evaluating a given historical query is also likely to increase monotonically over time. Furthermore, as only the most recent historical data in the current state of a historical or temporal relation is likely to change between accesses, there is likely to be an increasing amount of redundant processing associated with each

repeated evaluation of a historical query. As we discussed in the previous chapter, whether re-evaluation of a recurring historical query each time it is asked is cost-effective depends on application-specific factors such as the frequency of the query, update patterns, the cost of each evaluation, and the cost of alternate query processing techniques. Yet, there will be a subclass of recurring historical queries in many applications for which query re-evaluation each time a query is asked will have unacceptable cost. Also, the size of this subclass of recurring historical queries will increase during the life of a temporal database. Queries in this subclass, however, may be efficiently supported by implementing them as incrementally maintained materialized views.

We also emphasize incremental view materialization because it has been proven to be an appropriate strategy for query evaluation in several, diverse processing environments. As the work of Snodgrass, Horwitz, and Roussopoulos has shown, incremental view materialization is an appropriate strategy for query evaluation when query response time is a primary concern. Hence, TDBMS's need to support incremental view materialization to be of practical use in those processing environments where query response time is important.

In the next section we describe an architecture for query processing that accommodates incremental view materialization in TDBMS's. This architecture is based on the paradigm for incremental expression evaluation proposed by Snodgrass, Horwitz, and Roussopoulos. Then we describe our prototype query processor for TQuel, which uses an update network, constructed using this architecture, to update materialized views incrementally following changes to their underlying relations. We conclude the chapter with a discussion of optimization techniques for update networks containing nodes that implement incremental historical operators.

7.3 Architecture

In this section we describe an architecture for query processing that accommodates incremental view materialization in TDBMS's. This architecture is an extension of the conventional architecture for query processing shown in Figure 7.4. Here ovals represent processing phases, rectangles represent data structures, and arcs indicate the access to data structures required during each phase of query processing. In conventional query processing, a query passes through four phases of processing [Aho et al. 1986, Date 1986D]. The syntactic analyzer builds a parse tree for the query, which is then checked for correctness against the system catalog by the semantic analyzer. The parse tree, if semantically correct, is then passed to the code generator where it is optimized and mapped onto a query execution plan (i.e., a set of implementation procedures, one for each node in the optimized parse tree [Date 1986D]). The interpreter evaluates query execution plans using the graph reduction algorithm described in Section 7.1. Note that, in this architecture, queries are transient; their existence ends with their evaluation.

Figure 7.5 extends the conventional architecture for query processing to accommodate incremental maintenance of materialized views. The code generator is augmented to map

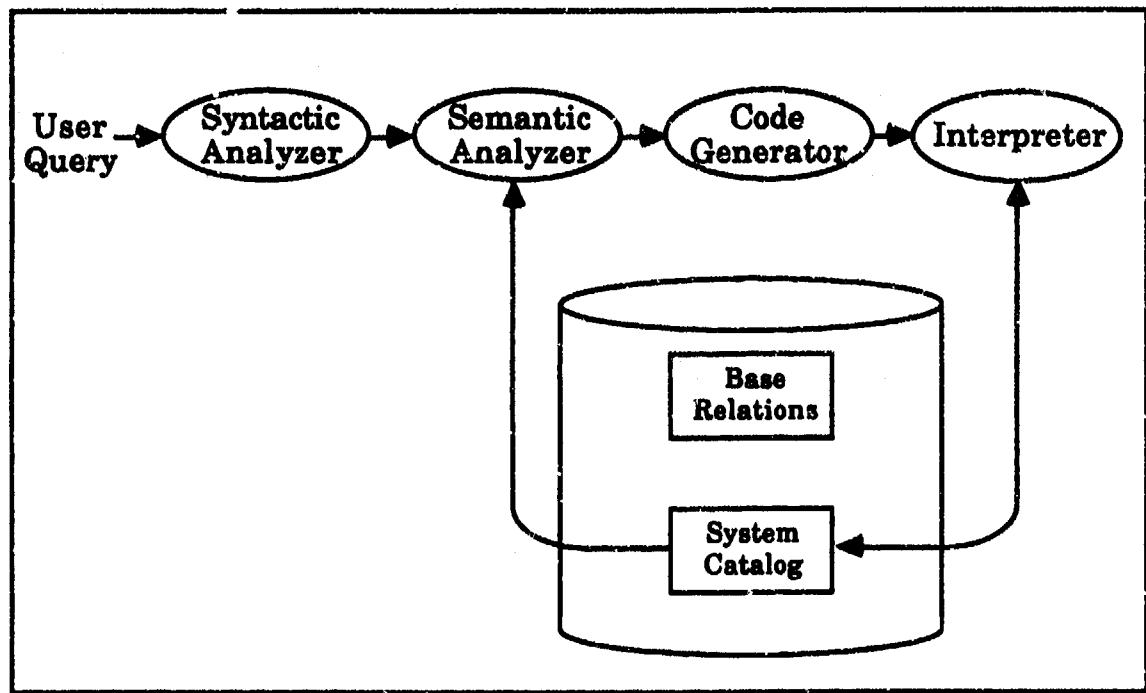


Figure 7.4: Conventional Architecture for Query Processing

the definitions of incrementally maintained materialized views onto update networks and to integrate these view update networks into a single update network for the database. The code generator also records each view's definition in the system catalog when the view is created. When a view is deleted, the code generator removes its definition from the system catalog. Also, if the view was being maintained incrementally, the code generator deletes its update network from the database's update network. The interpreter is augmented to activate the database's update network to update views incrementally whenever a base relation, upon which such views depend, is changed. Also, intermediate relation states for nodes in the network are stored between activations of the network. Note that, in this extended architecture, view update networks, unlike query execution plans, are persistent.

Because a temporal database may contain snapshot, rollback, historical, and temporal relations, both snapshot and historical views are supported in this extended architecture. The definitions of materialized snapshot views are mapped onto persistent update networks, as formalized by Snodgrass and Horwitz, while the definitions of materialized historical views are mapped onto persistent update networks containing nodes that implement incremental historical operators rather than incremental snapshot operators. To provide the update networks access to the database, we introduce four additional node types. These node types implement the functions *S_Differential*, *H_Differential*, *S_Update*, and *H_Update* defined in the previous chapter. A differential node always is associated with a base relation. It appears as a root node in an update network and computes a differential whenever its base relation is changed. An update node always is associated with a view. It appears as a leaf node in an update network and updates its view to reflect each change to

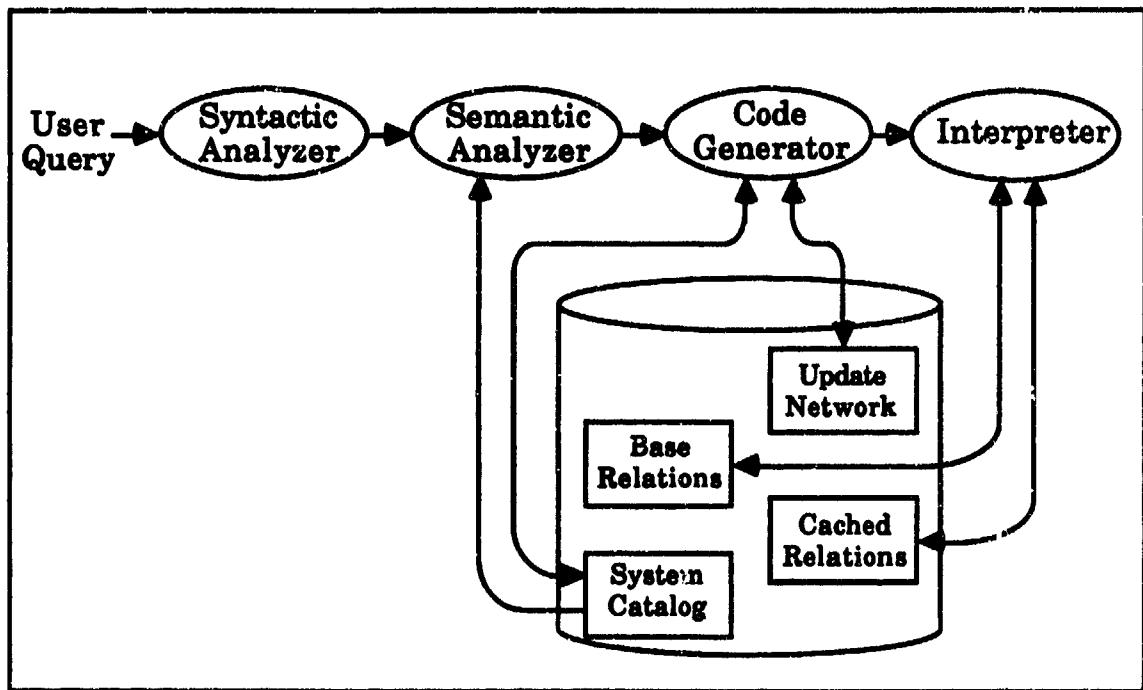


Figure 7.5: Extended Architecture for Query Processing

one of the view's underlying relations. Access to the database is restricted to differential and update nodes; operator nodes never access the database.

EXAMPLE. Once again, let S_1 denote a snapshot relation with attributes $\{sname, course\}$, S_2 denote a snapshot relation with attributes $\{hname, state\}$, and S_3 be a view defined by the following command.

```
define_incremental_view(S3, π(sname, state)(σsname=hname (S1×S2)))
```

Then, S_3 's update network is shown in Figure 7.6. Note that this update network differs from that shown in Figure 7.2 only in that differential and update nodes have been added to provide the network access to the database. Note also that if S_1 and S_2 had denoted historical relations rather than snapshot relations, the update network for S_3 would have been specified simply by replacing each node in Figure 7.6 with its historical counterpart. □

The update networks for all materialized views defined on a temporal database together form a *database update network*. Within a database update network, the update networks of individual views may be integrated to allow both node sharing among snapshot views and node sharing among historical views. A database update network in our architecture is analogous to Roussopoulos' *Logical Access Path Schema* [Roussopoulos 1982A].

EXAMPLE. Assume, as in the previous chapter, that S denotes a snapshot relation, whose current signature specifies the attributes $\{sname, course\}$, and that SP , SM , and SU are views that depend on S .

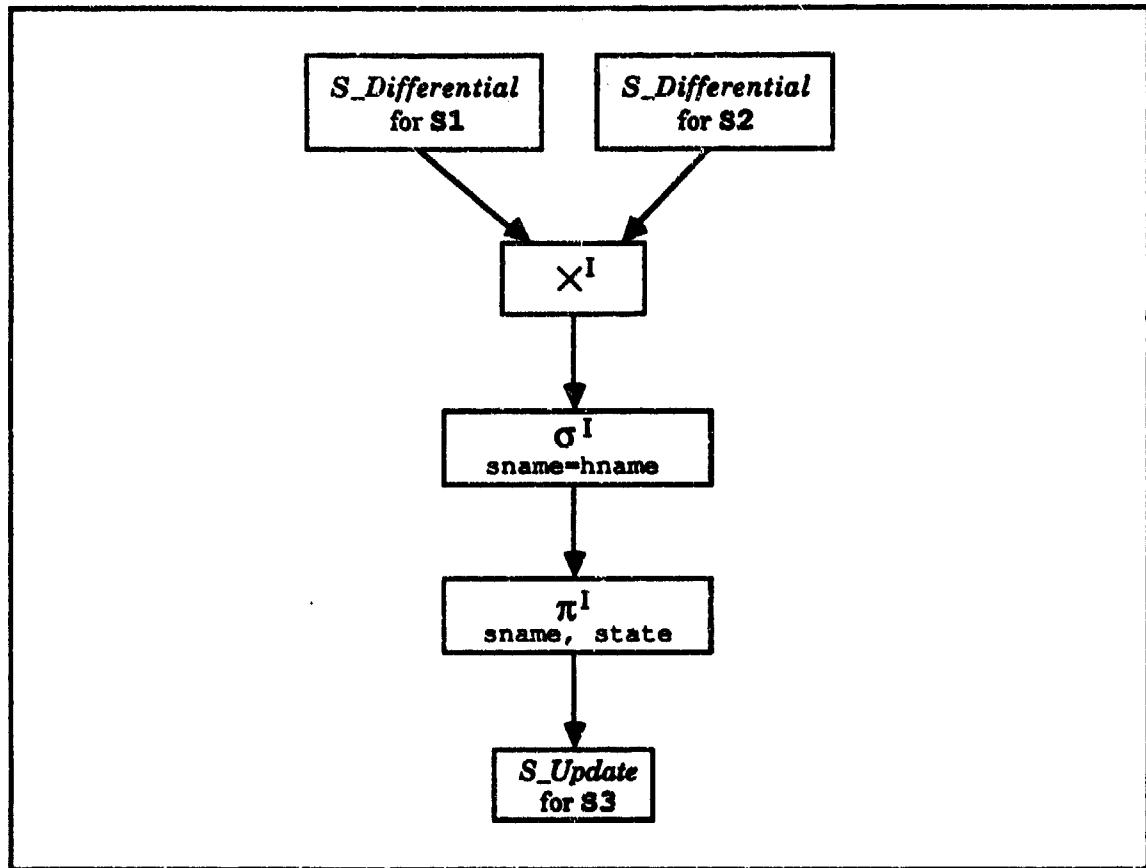


Figure 7.6: View Update Network for View S3

```

define_incremental_view(SP, σsname="Phil"(S))
define_incremental_view(SM, σsname="Marilyn"(S))
define_incremental_view(SU, π(sname)(SP ∪ SM))
  
```

If we also assume that these are the only views defined on the database containing S, then Figure 7.7 shows the update network for the database. Note that the update networks for SP and SU share nodes as do the update networks for SM and SU. \square

The previous two examples illustrate some important properties of a database update network, as we define it. First, there is exactly one differential node in the network for each base relation upon which at least one materialized view depends. Similarly, there is exactly one update node in the network for each materialized view. Second, all root nodes are differential nodes, all leaf nodes are update nodes, and all interior nodes are operator nodes. Third, the in-degree of nodes is fixed. The in-degree of differential nodes is 0, the in-degree of update nodes is 1, and the in-degree of each operator node is either

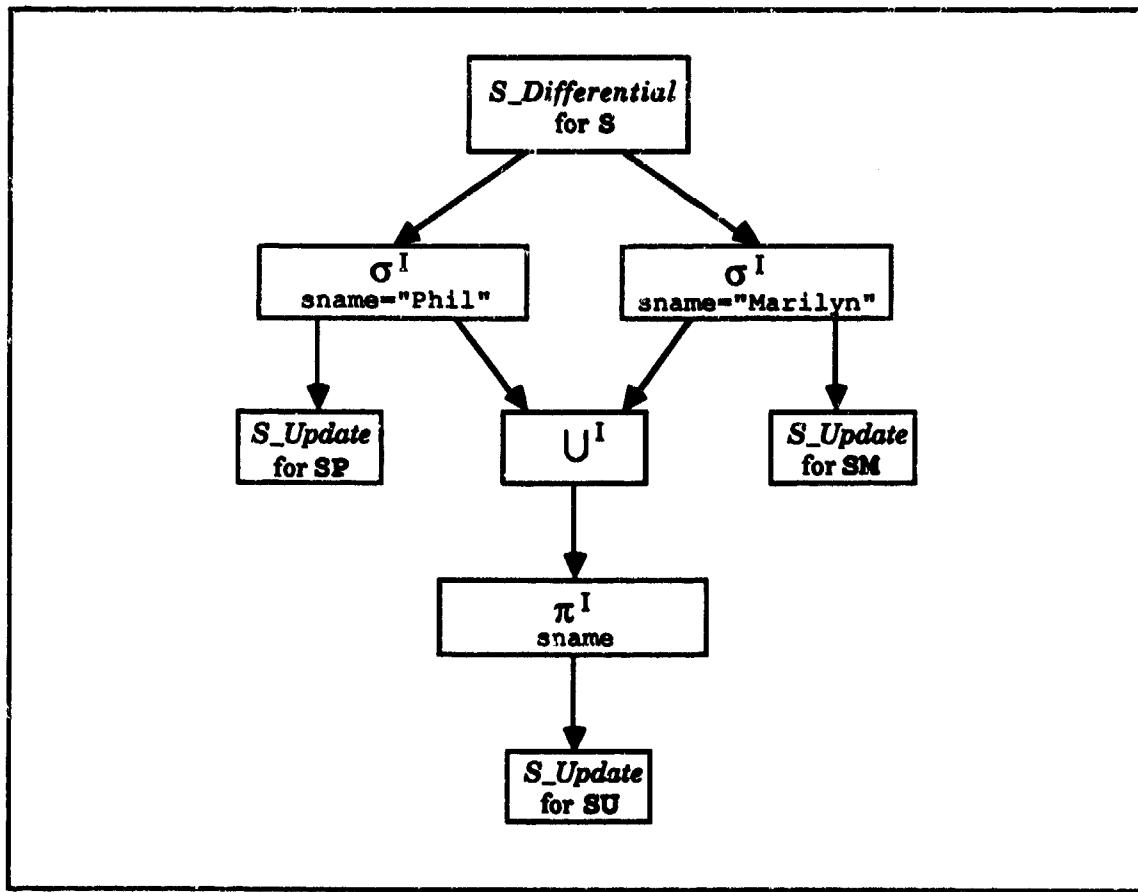


Figure 7.7: Database Update Network

1 or 2, depending on whether the node implements a unary or binary operator. Finally, the out-degree of update nodes is 0, but the out-degree of all other nodes is only required to be at least 1. Node sharing among view update networks determines the out-degree of differential and operator nodes.

7.4 TQuel Prototype

To show that our architecture is adequate for incremental maintenance of materialized historical views, we built a prototype query processor for TQuel. In this prototype, a database update network, defined in terms of incremental historical operators, is used to update materialized views incrementally following changes to their underlying relations. The prototype consists of two components: a code generator that maintains the database update network and an interpreter for a restricted subset of TQuel queries. We performed syntactic and semantic analysis of the TQuel queries by hand, since these aspects of query processing were not of interest to us. The prototype is written in C using the Interface Description Language (IDL) [Snodgrass 1988] for the specification of data structures. All data structures, including relations, materialized views, view definitions, and view update

networks are stored in main memory. The prototype supports creation, modification, and deletion of base relations; creation and deletion of materialized historical views; and execution of standard TQuel queries, not containing aggregates. Although we built the prototype to confirm that the architecture described in the previous section is adequate for incremental maintenance of materialized historical views in TDBMS's, building the prototype also provided us insight into several implementation issues, which we discuss in Section 7.5.

7.4.1 The Code Generator

The code generator maintains the database update network. Whenever it encounters a `define_incremental_view` command, it adds the update network for the view to the database update network. Likewise, whenever it encounters a `destroy` command for a materialized view, it removes the view's update network from the database update network.

In the prototype, views may be defined as any standard TQuel query, not containing aggregates, whose `as of` clause defaults to "now." TQuel queries with an `as of` clause other than "now" are rollback queries. Because past states of rollback and temporal relations never change, a rollback query always produces the same result and, hence, offers no insight to incremental view materialization. Theorem 5.1 on page 104 shows that a TQuel query that satisfies the above restrictions is equivalent to the algebraic expression

$$\pi_X(\delta_G, \{(I_{1,1}, V_{1,1}), \dots, (I_{k,m_k}, V_{k,m_k})\})(\delta_F(I_1 \hat{\times} \dots \hat{\times} I_k))$$

where, I_1, \dots, I_k denote relations; $I_{1,1}, \dots, I_{k,m_k}$ denote the attributes of those relations; F and G are boolean functions for non-temporal and temporal selection, respectively; $V_{1,1}, \dots, V_{k,m_k}$ are temporal functions; and X is the set of projection attributes. The code generator maps view definitions of this form onto update networks of the form shown in Figure 7.8. The code generator constructs the update network for all views using this basic structure; it doesn't attempt to tailor a view's update network for its efficient execution. Also, in adding a view's update network to the database update network, the code generator doesn't attempt to share operator nodes in the update networks of existing views. Node sharing is limited to differential nodes. We discuss techniques for optimizing update networks and implementing node sharing in Section 7.5.1.

Nodes in the database update network are implemented as IDL data structures. Nodes contain pointers to their ancestor(s) and information about each of their descendants in the network. Also, operator nodes contain data structures for caching their input relation state(s) between activations of the network. Physical copies of these intermediate relation states are stored as unstructured sets; the prototype doesn't implement any techniques for efficient caching of intermediate relation states. In Section 7.5.2 we discuss the applicability of the strategies, proposed by Borowitz and Roussopoulos, for efficient caching of intermediate relation states in snapshot-view update networks to historical views. In addition to the information common to nodes, each operator node contains information particular to the incremental historical operator it implements (e.g., nodes that implement the selection operator contain a semantically analyzed parse tree for their selection predicate).

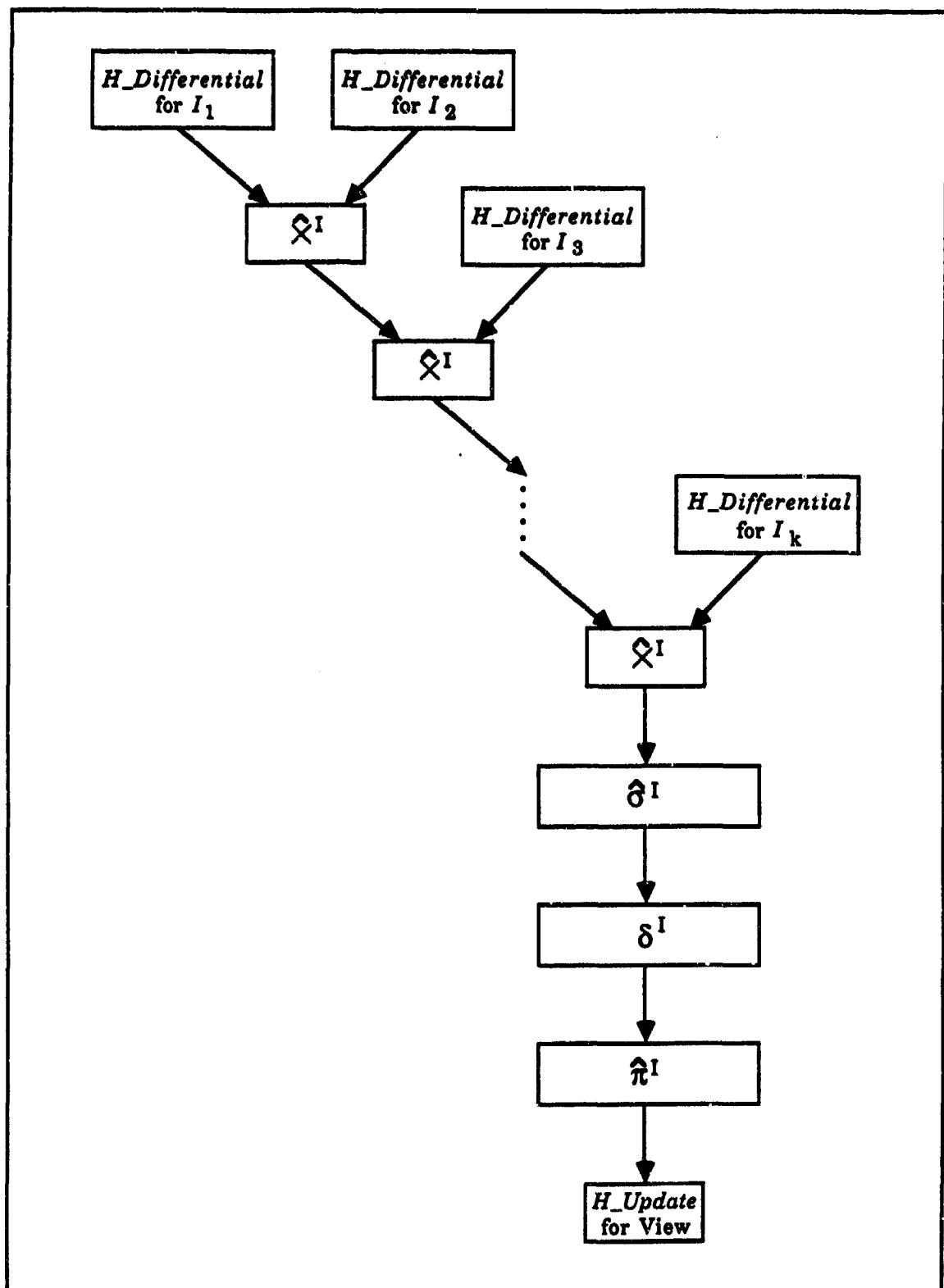


Figure 7.8: Update Network for a TQuel View

7.4.2 Interpreter

The interpreter executes a restricted subset of TQuel queries. The prototype supports the standard TQuel retrieve statement, without aggregates, for display of query results only. It also supports versions of the append and delete statements, restricted to the insertion, modification, and removal of a single tuple from a base relation. On each change to a base relation, the interpreter activates the database update network, which causes the differential for the change to be propagated through the network in depth-first order. Differentials are sets of before and after images of tuples as defined in the previous chapter. For each operator node on a path from the base relation's differential node to a view's update node, a procedure is called with a differential as a parameter. The procedure performs two functions. It uses the input differential and the node's cached input relation state(s) to compute the node's output differential. It then uses the input differential to update the node's cached input relation state(s) for the next activation of the network. A single copy of the procedure that performs these functions for a node type is shared by all the nodes in the network of that type.

The interpreter supports immediate update, but not deferred update, of materialized views. Also, it allows only the sequential processing of changes to base relations through the update network, and it implements no recovery procedures for update networks. We discuss extensions of the prototype to support deferred view materialization in Section 7.5.7. Concurrency control and recovery are discussed in Section 7.5.8.

7.5 Implementation Issues

As we stated earlier, building an update network that is correct addresses only one aspect of the implementation problem. To be of practical use, the network also must be efficient. Hence, in this section, we examine the applicability of existing optimization techniques, to be used to improve the performance of update networks for snapshot views, to update networks for historical views. We also discuss optimization techniques, to be used to improve the performance of update networks for historical views, that have no snapshot counterpart. In so doing, we argue that historical-view update networks are as amenable to efficient implementation as are update networks for snapshot views.

7.5.1 Query Optimization

Because view definitions are simply algebraic expressions, a view is analogous to a stored query that is re-executed after each change to one of its underlying relations and an update network is analogous to a query plan. Hence, the strategies for both local and global query optimization can be applied to update networks. We first consider the applicability of techniques for local query optimization to the update network of a single historical view and then consider the applicability of techniques for global query optimization to a database update network.

Local Optimization

Local query optimization concerns the problem of selecting the most efficient query plan for a query from the set of all its possible query plans. This problem for snapshot queries has been studied extensively and heuristic algorithms for selection of a near optimal query plan based on a statistical description of the database and a cost model for query plan execution have been proposed [Hall 1976, Jarke & Koch 1984, Krishnamurthy et al. 1986, Selinger et al. 1979, Smith & Chang 1975, Stonebraker et al. 1976, Wong & Youssef 1976, Yao 1979].

One important aspect of local query optimization is the transformation of one query plan into an equivalent, but more efficient, query plan. The size of the search space of equivalent query plans for a snapshot query is determined in part by the algebraic equivalences available in the snapshot algebra. Both Ullman and Maier identify equivalences that are available in the snapshot algebra for query plan transformation and describe their usefulness to query optimization [Maier 1983, Ullman 1982]. We show here that all but one of the equivalences that hold for the snapshot algebra also hold for the historical algebra defined in Chapter 3. In addition, we identify equivalences for the historical algebra that involve the historical derivation operator. Because all but one of the equivalences that hold for the snapshot algebra also hold for our historical algebra, the search space of equivalent query plans for a historical query should be comparable in size to that for an analogous snapshot query. Hence, our historical algebra does not limit the practical use of query plan transformation as an optimization technique for historical queries. Also, most algorithms for optimization of snapshot queries may be extended to optimize historical queries by taking into account the possible presence of historical derivation operators in query plans.

Our historical algebra supports all but one of the commutative, associative, and distributive equivalences involving only union, difference, and cartesian product in set theory [Enderton 1977]. The algebra does not support the distributive property of cartesian product over difference. (We argue in Chapter 8 that this equivalence is not a desirable property of historical algebras). The algebra also supports all the non-conditional commutative and distributive laws involving selection and projection presented by Ullman [Ullman 1982]. Finally, the algebra supports the commutative law of historical selection and historical derivation. For the theorems that follow assume that Q , R , and S are historical relation states.

Theorem 7.1 *The following equivalences hold for the historical algebra defined in Chapter 3.*

$$Q \cup R \equiv R \cup Q \quad (1)$$

$$Q \hat{\times} R \equiv R \hat{\times} Q \quad (2)$$

$$\sigma_{F_1}(\sigma_{F_2}(Q)) \equiv \sigma_{F_2}(\sigma_{F_1}(Q)) \quad (3)$$

$$\delta_{G, \{(I_1, V_1), \dots, (I_{m_q}, V_{m_q})\}}(\hat{\sigma}_F(Q)) \equiv \hat{\sigma}_F(\delta_{G, \{(I_1, V_1), \dots, (I_{m_q}, V_{m_q})\}}(Q)) \quad (4)$$

$$Q \dot{\cup} (R \dot{\cup} S) \equiv (Q \dot{\cup} R) \dot{\cup} S \quad (5)$$

$$Q \dot{\times} (R \dot{\times} S) \equiv (Q \dot{\times} R) \dot{\times} S \quad (6)$$

$$Q \dot{\times} (R \dot{\cup} S) \equiv (Q \dot{\times} R) \dot{\cup} (Q \dot{\times} S) \quad (7)$$

$$\partial_F(Q \dot{\cup} R) \equiv \partial_F(Q) \dot{\cup} \partial_F(R) \quad (8)$$

$$\partial_F(Q \dot{-} R) \equiv \partial_F(Q) - \partial_F(R) \quad (9)$$

$$\#_X(Q \dot{\cup} R) \equiv \#_X(Q) \dot{\cup} \#_X(R) \quad (10)$$

PROOF. The proofs of the first two equivalences follow directly from the definitions of historical union and historical cartesian product given in Chapter 3. For the third equivalence, consider the left-hand side of the equivalence. From the definition of historical selection on page 28, we have that a tuple q is in $\partial_{F_1}(\partial_{F_2}(Q))$ if, and only if, $F_1(q) \wedge q \in \partial_{F_2}(Q)$, which implies that q is in $\partial_{F_1}(\partial_{F_2}(Q))$ if, and only if, $F_1(q) \wedge F_2(q) \wedge q \in Q$. Now consider the right-hand side of the equivalence. Again from the definition of historical selection on page 28, we have that a tuple q is in $\partial_{F_2}(\partial_{F_1}(Q))$ if, and only if, $F_2(q) \wedge q \in \partial_{F_1}(Q)$, which implies that q is in $\partial_{F_2}(\partial_{F_1}(Q))$ if, and only if, $F_2(q) \wedge F_1(q) \wedge q \in Q$. Hence, the two expressions are shown to denote the same relation state. Proofs for the other equivalences, although more notationally cumbersome, can be constructed in a similar fashion. ■

Theorem 7.2 *The distributive property of cartesian product over difference does not hold for the historical algebra defined in Chapter 3.*

$$Q \dot{\times} (R \dot{-} S) \not\equiv (Q \dot{\times} R) \dot{-} (Q \dot{\times} S)$$

PROOF. We give an example when the equality does not hold. Let H_1 denote a historical relation whose current signature specifies the attributes {`hname`, `state`} and H_2 and H_3 denote historical relation whose current signature specifies the attributes {`sname`, `course`}. Furthermore, assume that their current states are as follows.

$$H_1 = \{ \langle ("Norman", \{1, 2, 3\}), ("Texas", \{1, 2, 3\}) \rangle \}$$

$$H_2 = \{ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle \}$$

$$H_3 = \{ \langle ("Norman", \{2\}), ("English", \{2\}) \rangle \}$$

Then,

$$H_1 \dot{\times} (H_2 \dot{-} H_3) = \{ \langle ("Norman", \{1, 2, 3\}), ("Texas", \{1, 2, 3\}), ("Norman", \{1\}), ("English", \{1\}) \rangle \}$$

$$(H_1 \dot{\times} H_2) \dot{-} (H_1 \dot{\times} H_3) = \{ \langle ("Norman", \emptyset), ("Texas", \emptyset), ("Norman", \{1\}), ("English", \{1\}) \rangle \}$$

Hence, $H_1 \dot{\times} (H_2 \dot{-} H_3) \neq (H_1 \dot{\times} H_2) \dot{-} (H_1 \dot{\times} H_3)$. ■

Ullman identifies several conditional equivalences involving selection and projection that can be used in optimizing snapshot queries [Ullman 1982]. These conditional equivalences also hold in our historical algebra (again, the proofs are cumbersome and unenlightening). We list these equivalences here, along with their accompanying conditions.

- If the non-temporal predicate F references only attributes of Q , then $\delta_F(Q \hat{\times} R) \equiv \delta_F(Q) \hat{\times} R$.
- If F can be expressed as $F_1 \wedge F_2$, where F_1 references only attributes of Q and F_2 references only attributes of R , then $\delta_F(Q \hat{\times} R) \equiv \delta_{F_1}(Q) \hat{\times} \delta_{F_2}(R)$.
- If F_1 references only attributes of Q but F_2 references attributes of Q and R , then $\delta_F(Q \hat{\times} R) \equiv \delta_{F_2}(\delta_{F_1}(Q) \hat{\times} R)$.
- If F references only attributes in the set X of projection attributes, then $\pi_X(\delta_F(Q)) \equiv \delta_F(\pi_X(Q))$.
- If F also references attributes X' that are not in the set X of projection attributes, then $\pi_X(\delta_F(Q)) \equiv \pi_X(\delta_F(\pi_{X \cup X'}(Q)))$.
- If X_1 and X_2 are sets of projection attributes where $X_1 \subseteq X_2$, then $\pi_{X_1}(\pi_{X_2}(Q)) \equiv \pi_{X_1}(Q)$.
- If X is a set of projection attributes where X_q are attributes of Q , X_r are attributes of R , and $X_q \cup X_r = X$, then $\pi_X(Q \hat{\times} R) \equiv \pi_{X_q}(Q) \hat{\times} \pi_{X_r}(R)$.

In addition to the conditional equivalences involving selection and projection, several conditional equivalences involving historical derivation, which have no snapshot counterparts, hold for the historical algebra. For these equivalences, recall from the definition of historical derivation on page 34 that

$$\delta_G, \{(I_1, I_1), \dots, (I_{m_q}, I_{m_q})\}(Q)$$

is a special form of the derivation operator that performs only the temporal selection function. Because this special form of historical derivation has properties analogous to those of non-temporal selection, the following equivalences involving historical derivation hold.

$$\begin{aligned} \delta_G, \{(I_1, V_1), \dots, (I_{m_q}, V_{m_q})\}(\delta_G, \{(I_1, I_1), \dots, (I_{m_q}, I_{m_q})\}(Q)) &\equiv \delta_G, \{(I_1, V_1), \dots, (I_{m_q}, V_{m_q})\}(Q) \\ \delta_{G_1}, \{(I_1, I_1), \dots, (I_{m_q}, I_{m_q})\}(\delta_{G_2}, \{(I_1, I_1), \dots, (I_{m_q}, I_{m_q})\}(Q)) &\equiv \\ \delta_{G_2}, \{(I_1, I_1), \dots, (I_{m_q}, I_{m_q})\}(\delta_{G_1}, \{(I_1, V_1), \dots, (I_{m_q}, V_{m_q})\}(Q)) & \end{aligned}$$

If the temporal predicate G references only attributes of Q , then

$$\delta_G, \{(I_{q,1}, I_{q,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}(Q \hat{\times} R) \equiv \delta_G, \{(I_{q,1}, I_{q,1}), \dots, (I_{q,m_q}, I_{q,m_q})\}(Q) \hat{\times} R.$$

If G can be expressed as $G_1 \wedge G_2$, where G_1 references only attributes of Q and G_2 references only attributes of R , then

$$\delta_{G, \{(I_{q,1}, I_{q,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}}(Q \hat{\times} R) \equiv \\ \delta_{G_1, \{(I_{q,1}, I_{q,1}), \dots, (I_{q,m_q}, I_{q,m_q})\}}(Q) \hat{\times} \delta_{G_2, \{(I_{r,1}, I_{r,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}}(R).$$

If G_1 references only attributes of Q but G_2 references attributes of Q and R , then

$$\delta_{G, \{(I_{q,1}, I_{q,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}}(Q \hat{\times} R) \equiv \\ \delta_{G_2, \{(I_{q,1}, I_{q,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}}(\delta_{G_1, \{(I_{q,1}, I_{q,1}), \dots, (I_{q,m_q}, I_{q,m_q})\}}(Q) \hat{\times} R).$$

These conditional equivalences involving historical derivation are important because they can be used to move temporal selection before cartesian product in a query plan transformation. The above equivalences imply that if G can be expressed as $G_1 \wedge G_2$, where G_1 references only attributes of Q and G_2 references only attributes of R , then

$$\delta_{G, \{(I_{q,1}, V_{q,1}), \dots, (I_{r,m_r}, V_{r,m_r})\}}(Q \hat{\times} R) \equiv \delta_{G, \{(I_{q,1}, V_{q,1}), \dots, (I_{r,m_r}, V_{r,m_r})\}}(\\ \delta_{G_1, \{(I_{q,1}, I_{q,1}), \dots, (I_{q,m_q}, I_{q,m_q})\}}(Q) \hat{\times} \delta_{G_2, \{(I_{r,1}, I_{r,1}), \dots, (I_{r,m_r}, I_{r,m_r})\}}(R)).$$

Performing the temporal selection function *twice* may be cost effective, depending on the size of Q and R and the selectivity of the predicates G_1 and G_2 .

Note that no equivalences are presented that involve historical derivation and union, difference, or projection. Historical derivation doesn't commute with projection or distribute over union or difference, even conditionally, as these operators may change attribute time-stamps.

In summary, all the above non-conditional and conditional equivalences can be used, along with statistical descriptions of historical databases and cost models for query plan execution, to optimize individual historical queries.

Global Optimization

Global query optimization concerns the problem of integrating a set of query plans into a single plan that minimizes the cost of executing all the individual plans. Optimization of the different query plans individually does not necessarily ensure optimal overall query processing because it does not consider the potential for savings due to sharing of common subexpressions among queries [Roussopoulos 1982B]. Hence, identification of common subexpressions among queries is a central issue in global query optimization [Chakravarthy & Minker 1986]. Although global query optimization has not been studied as extensively

as local query optimization, algorithms for recognizing common subexpressions in multiple query plans have been developed and strategies for integrating a set of query plans into a single plan have been proposed [Chakravarthy & Minker 1986, Finkelstein 1982, Jarke & Koch 1984, Roussopoulos 1982A, Roussopoulos 1982B, Roussopoulos & Yeh 1984, Satoh et al. 1985, Sellis & Shapiro 1985].

Global query optimization allows sharing of common subexpressions among queries, which may produce saving even when only a few queries are considered. Chakravarthy has shown that there is a reasonable probability that, among a group of independently generated queries, references to base relations will overlap, even when as few as five queries are considered [Chakravarthy & Minker 1982]. Global query optimization, however, can be costly and may not be necessarily cost effective when a few queries are considered. Because database update networks are likely to support considerably more than five materialized views, there may be significant potential for node sharing. Hence, the benefits to be gained from optimizing a database update network are likely to justify the cost of executing a heuristic global optimization algorithm. Also, because database update networks are persistent, the cost of executing the algorithm can be amortized across multiple activations of the network.

Additional benefits may be gained by using global query optimization algorithms to maintain the database update network dynamically as views are created and destroyed and to identify materialized views and relation states cached at operator nodes that may be used to answer ad hoc queries efficiently. This latter task represents simply another opportunity for node sharing that can be exploited through global query optimization. In related work, Larson and Yang have studied the problem of mapping queries on base relations onto queries on views when the views, but not the base relations, are materialized [Larson & Yang 1985, Yang & Larson 1987].

As shown in Figure 7.7 our architecture for incremental view materialization accommodates node sharing among the update networks of different views. Also, the algebraic expression for a TQuel query is structurely similar to that of the frequently studied Projection-Selection-Join-expression in the snapshot algebra, and the historical operators all have properties similar to those of their snapshot counterparts. Hence, most algorithms for global optimization of snapshot queries may be extended to optimize a set of historical queries by taking into account the possible presence of historical derivation operators in query plans.

7.5.2 Local Storage Strategies at Operator Nodes

In the TQuel prototype, only differentials are passed among nodes. Each input relation state for an operator node in the database update network is cached at that node between activations of the network. Physical copies of these intermediate relation states are stored as sets with no consideration for efficiency. Yet, numerous techniques are available for efficient cacheing of intermediate relation states between activations of update networks. We discuss some of those efficiency techniques here. We also consider the applicability

of the techniques, proposed by Horwitz and Roussopoulos, for cacheing of intermediate relation states to temporal-database update networks.

In the previous chapter, both historical selection and historical derivation are defined in terms of their input differentials alone. For these two operators, an output differential is computed from an input differential, independent of the operator's input relation state. Hence, intermediate relation states need not be cached for nodes that implement either of these two operators. Cacheing is required, however, for nodes that implement the other historical operators. For these nodes, the spectrum of conventional data structuring techniques is available for building access paths to tuples in the cached relation states. Also, data structures for cached relation states can be tailored to support the data access requirements of each node type, or even each individual node. For example, nodes that implement union and difference need an access path for efficient retrieval of a tuple's value-equivalent counterpart, if one exists, from a cached relation state. Similarly, nodes that implement projection need an access path for efficient retrieval of tuples in a cached relation state that match a given tuple on the projection attributes. Data structures that accommodate efficient selective retrieval of tuples from snapshot relation states have been studied extensively; they accommodate, equally well, selective retrieval of tuples from historical relation states [Date 1986D, Ullman 1982].

Although each input relation state for a node that implements a historical operator, other than selection or derivation, needs to be cached between activations of the update network, these states need not necessarily be cached at the nodes to which they are input. Rather, it may be more appropriate sometimes to cache these intermediate relation states at the nodes from which they are output. For example, when global query optimization is used to construct a database update network, operator nodes may have an arbitrary number of children, where the number of children is determined by the number of view update networks that share the node. If a node has multiple children, it may be more efficient to cache the node's output relation state at that node rather than to cache a copy of the relation state as an input relation state at each of the node's children.

Cacheing the output relation state of an operator node, even if it is not needed as input to any of the node's children, may also sometimes be cost-effective. If dynamic global query optimization is used to integrate the update networks for new views into the existing database update network, access to the output relation state of the leaf node in a subnetwork that can be shared may aid in initializing the view whose update network is being added. Also, cacheing of the output relation states of certain operator nodes may aid in recovery by reducing the effort required to restore the network following a failure. Finally, cacheing the output relation state of a node that implements historical derivation may be cost-effective, even if it is not otherwise needed, because the processing time of the node can be reduced, if the node's output relation state is available (c.f., Section 7.5.5).

The approach proposed by Horwitz for implementing snapshot-database update networks in which cacheing of intermediate relation states is unnecessary, except for cartesian product nodes, can be extended to temporal-database update networks by implementing incremental historical operators using the *selective-retrieval*, rather than the *membership-*

test, function. As shown by Horwitz, most incremental snapshot operators can be implemented using only the membership-test function [Horwitz 1985]. A node that implements an incremental snapshot operator needs to know only whether a tuple in its input differential is in its input relation state (c.f., Section 6.3.2), and a call to the membership-test function of the node's parent provides this information. A node that implements an incremental historical operator, unlike its snapshot counterpart, needs to know whether a tuple in its input differential has a *value-equivalent* counterpart in its input relation state (c.f., Section 6.4.2). Hence, simple set-membership tests on a node's input relation state(s), while adequate to implement incremental snapshot operators, are inadequate to implement their historical counterparts. The selective-retrieval function, however, can be extended to provide the needed information. Rather than return tuples in a relation state that match values specified for some subset of attributes, selective-retrieval could be defined to return tuples that match the values specified for the *value-component* of some subset of attributes. Then, a node could call the selective-retrieval function of its parent to determine whether a tuple in its input differential had a value-equivalent counterpart in its input relation state and, if so, to return that tuple.

The approach proposed by Roussopoulos for implementing snapshot-database update networks, in which intermediate relation states are cached using indexing, also can be extended to temporal-database update networks. The following changes are necessary for the approach to work for temporal-database update networks.

- The output relation state of a union or difference node, which is cached as a vector of addresses in snapshot-database update networks, is cached as a two-dimensional *matrix of address pairs* in temporal-database update networks. Each pair of addresses points to value-equivalent tuples in the node's input relation states that contribute to a single tuple in its output relation state. One of the addresses is NIL if a tuple in one of the input relation states doesn't have a value-equivalent counterpart in the other relation state. This complication arises because two value-equivalent snapshot tuples are, by definition, identical, whereas two value-equivalent historical tuples need not, and most likely will not, be identical.
- The output relation state of a projection node, which is cached as a vector of addresses in snapshot-database update networks, is cached as a vector of *sets*, where each set contains the addresses of tuples in the node's input relation state whose attribute value-components match on the projection attributes. This complication arises because the projections of two tuples, if value-equivalent, need not be identical. Whereas two snapshot tuples that match values on the projection attributes contribute exactly the same information to the projection, two historical tuples, even though they match value-components on the projection attributes may contribute different temporal information to the projection. Note that although the addresses of those tuples in the node's input relation state whose contribution of temporal information to an output tuple is subsumed by that of other tuples need not be included in the cache, identification of such tuples may not be cost-effective.

- The output relation state of a historical derivation node, which has no counterpart in a snapshot-database update network, is cached here as a vector of addresses, where each address points to a tuple in the node's input relation state that is mapped onto a tuple in the node's output relation state.

The cacheing of the output relation state of a selection node as a vector of addresses and the output relation state of a cartesian product node as a two-dimensional matrix of address pairs need not be changed. The snapshot and historical versions of each of these operators perform the same function, only on snapshot and historical tuples, respectively.

The approaches proposed by Horwitz and Roussopoulos for cacheing intermediate relation states in database update networks can also be combined, with or without variations, to form hybrid approaches for cacheing intermediate relation states in database update networks. We present here only one such approach. In this approach, the following rules are used to cache intermediate relation states between activations of a temporal-database update network.

- Intermediate relation states for selection and derivation nodes are not cached; these nodes are memoryless.
- Union and difference nodes are implemented using selective-retrieval functions to eliminate the need to cache intermediate relation states for these nodes.
- The output relation states of cartesian product nodes are cached as vectors rather than two-dimensional matrices. Each element in the vector is a sequence of addresses, where the addresses point directly to the subtuples (e.g., base relation tuples) that make up a tuple in the relation state.
- The output relations states of projection and derivation nodes are cached in materialized form.

This approach takes advantage of the fact that intermediate relation states for selection and derivation nodes need not be cached. Union and difference are implemented using selective-retrieval functions because doing a union or difference operation on two value-equivalent tuples whenever a tuple in a union or difference node's output relation state is accessed may be more cost-effective than storing that relation state in materialized form. Cacheing the output relation states of cartesian product nodes as vectors, where each element in the vector is a sequence of addresses, allows tuples in those relation states to be materialized via a single level of indirection. Hence, in this approach, we are able to save space by cacheing addresses rather than tuples at cartesian product nodes but do not have the problem, present in Roussopoulos's approach, of having to follow arbitrary levels of indirection to materialize a tuple. The output relation states of projection and derivation nodes are cached in materialized form because cacheing the nodes' output relation states in materialized form is likely to be more cost-effective than recomputing a tuple in those relation states each time it is accessed.

In summary, there are many techniques available for cost-effective cacheing of intermediate relation states between activations of a database update network. Some are complementary while others are mutually exclusive. The appropriateness of these techniques to the design of a cacheing system for a particular update network is application-specific, depending on factors such as processing environment (e.g., centralized or distributed), processing constraints, storage constraints, communication constraints, size of relations, selectivity of nodes, and stability of the network. Also, a cacheing system for an update network, once designed, can be changed dynamically to tune the performance of the network. The problems that arise in the design of a cacheing system, and their solutions, however, are similar for both snapshot-database and temporal-database update networks.

7.5.3 Representation of Attribute Time-stamps

In the TQuel prototype the valid-time component of each attribute in a tuple is stored as a sequence of temporally ordered intervals, where each interval is closed at its left endpoint and open at its right endpoint. The sequence denotes the minimal set of intervals that covers all the chronons in the attribute's valid-time component.

EXAMPLE. Assume that {2, 5, 6, 7, 19, 20, 21, 22, 23, 24} is the valid-time component of an attribute. Then, it would be stored as the sequence ([2, 3), [5, 8), [19, 25)). \square

This representation of the valid-time components of attributes as sequences of intervals has two benefits: it is a space-efficient representation and the implied temporal ordering of the intervals can be used to advantage when implementing the historical operators that manipulate attribute time-stamps (i.e., union, difference, projection, and historical derivation). Note also that it may be possible to share sequences or subsequences of intervals among attributes that have chronons in common.

7.5.4 Representation of Historical Differentials

In the previous chapter, a historical differential was defined as a set of before and after images of tuples rather than as a set of incremental positive and negative temporal changes to tuples. This definition of historical differentials simplified somewhat definition of the historical operators. Our definition of historical differentials also allowed the historical derivation operator to be defined as a function on an input differential alone. If differentials had been defined as incremental positive and negative temporal changes to tuples, historical derivation would have had to be defined as a function on both an input relation state and an input differential. Hence, implementation of memoryless historical derivation nodes in a database update network requires that differentials be as defined in the previous chapter.

Also, the cost of processing a differential at a cartesian product node is less using our differentials. When differentials are sets of before and after images of tuples, a cartesian product node computes the output differential for a change to a tuple in one of its input relation states simply by concatenating the before and after images of the tuple that

changes with each tuple in its other input relation state. Concatenation alone, however, is inadequate in the other case. Because cartesian product does not distribute over difference in the historical algebra (c.f., Section 7.5.1), concatenation of a tuple that represents an incremental negative temporal change to a tuple in one of a cartesian product node's input relation states with a tuple in its other input relation state does not produce a correct incremental negative change for a tuple in the node's output relation state.

EXAMPLE. Let H_1 denote a historical relation whose current signature specifies the attributes {`hname`, `state`}, H_2 denote a historical relation whose current signature specifies the attributes {`sname`, `course`}, and $\Delta_{H_1}^-$ denote an incremental negative differential for H_1 (i.e., information that is *removed* from H_1 by an update), where

$$H_1 = \{ \langle ("Norman", \{1, 2, 3\}), ("Texas", \{1, 2, 3\}) \rangle \}$$

$$H_2 = \{ \langle ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle \}$$

$$\Delta_{H_1}^- = \{ \langle ("Norman", \{2\}), ("Texas", \{2\}) \rangle \}.$$

If a cartesian product node that implements $H_1 \hat{\times} H_2$ were to simply concatenate tuples in $\Delta_{H_1}^-$ and H_2 , the incremental negative differential $\Delta_{H_1 \hat{\times} H_2}^-$ (i.e., the information that should be removed from $H_1 \hat{\times} H_2$ as a result of the update) would contain the single tuple

$$\{ \langle ("Norman", \{2\}), ("Texas", \{2\}), ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle \}$$

but, the correct value for $\Delta_{H_1 \hat{\times} H_2}^-$ is

$$\{ \langle ("Norman", \{2\}), ("Texas", \{2\}), ("Norman", \emptyset), ("English", \emptyset) \rangle \}.$$

If, however, we were to represent this incremental negative differential using before and after images of tuples, Δ_{H_1} would be

$$\{ \langle \langle ("Norman", \{1, 2, 3\}), ("Texas", \{1, 2, 3\}) \rangle, \langle ("Norman", \{2\}), ("Texas", \{2\}) \rangle \rangle \}$$

and $\Delta_{H_1 \hat{\times} H_2}$ would be

$$\{ \langle \langle ("Norman", \{1, 2, 3\}), ("Texas", \{1, 2, 3\}), ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle, \langle ("Norman", \{2\}), ("Texas", \{2\}), ("Norman", \{1, 2\}), ("English", \{1, 2\}) \rangle \rangle \}. \quad \square$$

Although defining differentials as sets of before and after images of tuples eliminates the need to cache intermediate results for historical derivation nodes between update network activations and allows cartesian product to be implemented efficiently, it also may

cause some inefficiencies. To minimize the flow of differentials through the database update network, nodes that implement union, difference, projection, and historical derivation should only propagate a before and after image pair (h_b, h_a) to their children if $h_b \neq h_a$. Implementation of this restriction on the flow of differentials, however, requires that these operators perform an equality check on each differential pair they output, which may be expensive if the valid-time components of attributes in a tuple's before and after images are complex, but similar. Hence, it might be more cost-effective not to perform the check or to perform only a partial check (e.g., comparing no more than a fixed number of intervals for each attribute of the two tuples). Checking output differentials for equality is another task performed at nodes that can be adjusted dynamically to maximize overall performance of the network.

7.5.5 Local Processing Strategies at Operator Nodes

In this section we discuss the time complexity of propagating a change to a single tuple through the nodes in a database update network and present some techniques for reducing processing costs at nodes. As we emphasize below, the cost of processing a historical differential at a selection or cartesian product node in a temporal-database update network is similar to that of processing an analogous snapshot differential at a selection or cartesian product node in a snapshot-database update network. The cost of processing a historical differential at other node types, however, depends primarily on the number of attributes in a tuple and the number of intervals in the valid-time components of attributes. Also, optimization techniques exist that can be applied at projection and historical derivation nodes to reduce their processing costs. For this discussion, we assume that a differential is the before and after image of a single tuple. Also, we consider only three time complexity metrics: the maximum number of intervals in an attribute time-stamp of a tuple in the differential, the maximum number of attributes in a tuple, and the maximum number of tuples in an input relation state for a node. We do not consider the cost of accessing tuples cached in intermediate relation states; this cost will depend strongly on the storage structure used to cache the tuples. Space complexity was discussed informally in Section 7.5.2.

Selection and Cartesian Product

The time complexity of processing a differential at a selection or cartesian product node is similar in temporal-database and snapshot-database update networks. The cost of processing a differential at a selection node depends only on the complexity of the selection predicate, while the cost of processing a differential for one of a cartesian product node's input relation states depends on the number of tuples in the node's other input relation state. Hence, a selection node has constant time complexity in the number of intervals in an attribute time-stamp, the number of attributes in a tuple, and the number of tuples in the input relation state. A cartesian product node has constant time complexity in the number of intervals and the number of attributes, but has linear time complexity in the number of tuples in an input relation state.

Union and Difference

Processing a differential at a union or difference node requires that the temporal union or difference be computed for, at most, two pairs of tuples. Furthermore, these calculations are only necessary if a differential for one of the node's input relation states has a value-equivalent counterpart in the node's other input relation state. The cost of doing a temporal union or difference of two tuples is the cost of performing union or difference operations on the valid-time components of the tuple's attributes. Because valid-time is represented as a temporally ordered sequence of intervals, the processing time for the temporal union or difference of two tuples depends on the number of intervals in the valid-time components of the attributes in the two tuples. Hence, union and difference nodes have linear time complexity in the number of intervals and the number of attributes, where their total processing costs depend on the product of the two. Both union and difference nodes have constant time complexity in the number of tuples.

Projection

If we assume that a projection node's input relation state is cached, processing a change to a tuple at a projection node requires that two historical projections be performed on the subset of tuples in the node's input relation state that matches the attribute value-components of the tuple being changed on the projection attributes, one before the change and the other after the change. In the worst case, two historical projections of the entire input relation state would be required, but only if all tuples in the input relation state matched the attribute value-components of the changed tuple on the projection attributes. Because historical projection performs a temporal union of the valid-time components of the projection attributes of all the qualifying tuples, the time required to process a differential at a projection node depends on the number of qualifying tuples, the number of projection attributes, and the number of intervals in the valid-time components of those attributes. Hence, projection nodes have linear time complexity in the number of intervals, number of projection attributes, and the number of tuples, where the processing costs depend on the product of the three.

There are at least three techniques that can be used to reduce processing costs at projection nodes. The obvious technique is to compute the historical projection on all the qualifying tuples, except the tuple being changed, once and reuse this temporary result, along with the before and after images of the tuple being changed, to compute the before and after images of the output differential.

The cost of processing a differential at a projection node also may be reduced by extending a technique proposed by Blakeley, Larson, and Tompa for efficient implementation of incremental snapshot projection [Blakeley et al. 1986A] to incremental historical projection. They propose that the output relation state of a snapshot projection node be cached and that a count be maintained for each tuple in the output relation state of the number of tuples in the node's input relation state that project onto that tuple. Then, whenever a tuple is added to the node's input relation state, its insertion is recorded in the cache. If

the tuple's projection is already in the cache, its reference count is incremented; otherwise, the projection is added to the cache with an initial reference count of one. Likewise, whenever a tuple is deleted from the node's input relation state, its deletion is recorded in the cache. If the tuple's projection in the cache has a reference count of one, the projection is physically removed from the cache; otherwise, its reference count is simply decremented.

This technique can be applied to nodes that implement historical projection, with one important change. The reference counts can't be associated with tuples; they must be associated with chronons in the valid-time components of attributes. In the snapshot algebra, a tuple in a projection node's output relation state may be the image under projection of an arbitrary number of tuples in the node's input relation state. Analogously, in the historical algebra, a chronon in the valid-time component of an attribute of a tuple in a projection node's output relation state may be the image under projection of a chronon in the valid-time component of the same attribute of an arbitrary number of tuples in the node's input relation state. A variation of the algorithm described above for snapshot projection can be used to process a differential at a historical projection node when the node's output relation state is cached and reference counts are maintained for chronons, rather than tuples.

Finally, the most cost-effective approach for implementing projection nodes may be the use of both techniques described above in combination. Under this hybrid approach, tuples in a projection node's input relation state would be cached and projections would be recomputed for each differential until the number of tuples in the node's input relation state that matched value components on the projection attributes reached a threshold, which could be fixed or dynamically set to manage the node's performance. Once the threshold had been reached, the projection of the qualifying tuples would be computed and cached, along with chronon reference counts, for use in processing future differentials for this subset of tuples.

Historical Derivation

Processing costs at historical derivation nodes, like those at projection nodes depend on the number of intervals in the valid-time components of attributes and the number of attributes in a tuple. To process the before or after image of an m -tuple, a historical derivation node must evaluate a temporal predicate G and temporal functions V_1, \dots, V_m for all possible assignments of intervals in valid-time components of attributes to their attributes' names. Hence, historical derivation nodes have, in the worst case, exponential time complexity in the number of attributes, polynomial time complexity in the number of intervals, and constant time complexity in the number of tuples. Note, however, that this time complexity for answering a query involving non-synchronized attributes (i.e., attributes whose values do not change simultaneously [Navathe & Ahmed 1987]) is the same as that in historical algebras where relations are restricted to synchronized attributes and tuples are time-stamps. In those algebras, a cascade of cartesian products is required to answer a query involving multiple non-synchronized attributes. The time complexity of a cascade of cartesian products is exponential in the number of cartesian products, which

is analogous to the number of non-synchronized attributes. Embedding the "temporal cartesian product" of attribute time-stamps in the historical derivation node, however, has an advantage over the use of cartesian product nodes to perform this task. Optimization strategies are more easily applied within the historical derivation operator than across cartesian product nodes to reduce processing costs.

Although historical derivation nodes have exponential time complexity in the number of attributes and polynomial time complexity in the number of intervals, there are several techniques for reducing this cost. These heuristics make the average-case cost substantially less than the worst-case cost. For example, not all attribute time-stamps need to be considered when evaluating a tuple. Only assignments of intervals to attribute names need be considered for those attributes whose names appear in either the temporal predicate G , or a temporal function V_1, \dots, V_m . We used this technique in our prototype TQuel processor to reduce the cost of performing historical derivations to reasonable levels. Also, if a temporal function V_j , $1 \leq j \leq m$, is defined by an expression that is simply an attribute name, that attribute need not be considered in the assignment of intervals to attribute names, unless the attribute name also appears in G or some other temporal function. If $V_j = I_k$, $1 \leq j, k \leq m$, which is common, the time-stamp of the attribute corresponding to V_j in the output tuple is simply the time-stamp of attribute I_k in the input tuple, if there is at least one assignment of intervals to attribute names that satisfies G , and the empty set, if there is no assignment of intervals to attribute names that satisfies G . These techniques alone will likely be sufficient to make the cost of processing a differential at most historical derivation nodes reasonable because most historical queries are likely to reference only a few attribute names in their predicate and temporal functions.

For those attributes that must be included in the assignment of intervals to attribute names, other heuristics are available for reducing further the number of combinations of assignments that must be considered. For example, the temporal predicate can be used, along with the temporal ordering of intervals in each attribute time-stamp, to limit the portion of the search space of assignments that must be considered. This technique provides the opportunity to reduce processing costs at some historical derivation nodes significantly.

Finally, less dramatic, but none the less effective, techniques for reducing processing costs are available. For example, the results of computing common subexpressions for the functions V_1, \dots, V_m can be shared. Also, if the output relation state at the historical derivation node is cached, the before image of the tuple in the output differential is available and, hence, requires no recomputation.

Composite Operator Nodes

In snapshot-database update networks, it may be cost-effective to combine two or more snapshot operations into a single composite operator node [Snodgrass 1982]. For example, combining selection and cartesian product into a single node may be beneficial. Analogies exists for temporal-database update networks. We consider one here. In Section 7.5.1, we presented algebraic properties of the historical algebra that can be used in optimizing

update networks. One useful property, the distributive property of historical derivation over cartesian product, does not hold, however, except in restricted cases. Even if G can be expressed as $G_1 \wedge G_2$, where G_1 references only attributes of Q and G_2 references only attributes of R ,

$$\delta_{G, \{(I_{q,1}, V_{q,1}), \dots, (I_{q,m_q}, V_{q,m_q})\}}(Q \hat{\times} R) \neq$$

$$\delta_{G_1, \{(I_{q,1}, V_{q,1}), \dots, (I_{q,m_q}, V_{q,m_q})\}}(Q) \hat{\times} \delta_{G_2, \{(I_{r,1}, V_{r,1}), \dots, (I_{r,m_r}, V_{r,m_r})\}}(R).$$

The property does not hold only because the expression on the left may produce a tuple whose attribute time-stamps for $I_{q,1}, \dots, I_{q,m_q}$ or for $I_{r,1}, \dots, I_{r,m_r}$ all are the empty set. The expression on the right disallows these tuples because a historical derivation operator, by definition, can't output a tuple whose attribute time-stamps all are the empty set. We can, however, effectively gain the benefits of this property by constructing a composite operator node that performs both historical derivation and cartesian product. In this node, a preprocessor for the node's left input would perform the function of $\delta_{G_1, \{(I_{q,1}, V_{q,1}), \dots, (I_{q,m_q}, V_{q,m_q})\}}(Q)$ and a preprocessor for the node's right input would perform the function of $\delta_{G_2, \{(I_{r,1}, V_{r,1}), \dots, (I_{r,m_r}, V_{r,m_r})\}}(R)$, with one exception. Both would pass output tuples whose attribute time-stamps are all the empty set to code that implements a slightly revised cartesian product operation. This code would perform the function of cartesian product, also with one exception. It would not output tuples whose attribute time-stamps all were the empty set, but would output tuples whose attribute time-stamps for either $I_{q,1}, \dots, I_{q,m_q}$ or for $I_{r,1}, \dots, I_{r,m_r}$ all were the empty set.

Table 7.1 summarizes the time complexity at historical operator nodes for processing single-element differentials.

7.5.6 Dynamic Time-stamps

Until now, we have only considered attribute time-stamps that contain intervals whose endpoints are fixed. Situations, however, often arise when it is appropriate to include, in an attribute's valid-time component, a dynamic interval (i.e., one whose right endpoint is not fixed but moves forward as time passes). For example, the time when an employee receives his current salary is often represented as a dynamic interval whose right endpoint is "now." We examine in this section the applicability of our architecture for incremental maintenance of materialized views to temporal databases in which time-stamps are allowed to contain a dynamically expanding interval.

When attribute time-stamps are allowed to contain dynamic intervals, expression evaluation requires that the right endpoint of such intervals be fixed for purposes of expression evaluation to allow for temporal comparisons and computations. The value assigned to a dynamic interval's right endpoint for expression evaluation depends on the meaning associated with that endpoint. For example, if dynamic intervals are assumed to extend

| Time Complexity | | | |
|-----------------|-----------------------|-----------|-------------|
| | Tuple | Attribute | Interval |
| Operators | Selection | Constant | Constant |
| | Cartesian Product | Linear | Constant |
| | Union | Constant | Linear |
| | Difference | Constant | Linear |
| | Projection | Linear | Linear |
| | Historical Derivation | Constant | Exponential |
| | | | Polynomial |

Table 7.1: Time Complexity of Incremental Historical Operators

"forever," then the appropriate value would be ∞ . If, however, dynamic intervals are assumed to extend only to "now," then the appropriate value would be the start time of the expression's evaluation, obtained from a system clock.

Our architecture accommodates time-stamps containing dynamic intervals, but not necessarily without performance penalty. If the value assigned the right endpoint of a dynamic interval is ∞ , dynamic intervals can be handled the same as fixed intervals without problem. If the value assigned the right endpoint of a dynamic interval is the start time of expression evaluation and post-active changes are *not* allowed (i.e., no chronon in an attribute time-stamp is greater than the chronon that denotes the start time of the expression's evaluation), dynamic intervals also pose no problem. If, however, the value assigned the right endpoint of a dynamic interval is the start time of expression evaluation and post-active changes are allowed, then a problem arises. A temporal predicate or a temporal constructor may produce a different result depending on when the expression is evaluated.

Example. Let H denote a historical relation whose current signature specifies the attributes `{hname, state}`. Assume that, for a tuple in H , the valid-time component of attribute `hname` is $[25, \text{now}]$ and the valid-time component of attribute `state` is $[45, 48]$. Now consider the temporal predicate `end of hname precede start of state`. If the predicate were evaluated at time 40, the predicate would be true. But, if the predicate were evaluated at time 45, or after, the predicate would be false. \square

Because expansion of a dynamic interval is implicit rather than explicit, differentials for these changes would not be generated at differential nodes in a database update network.

Hence, these implicit changes to dynamic intervals would not be propagated through the network to view update nodes. Yet, a view defined in terms of the temporal predicate on relation H in the above example would possibly change at time 45. A change, however, would not be recognized. There are two basic solutions to this problem. One possible, but inefficient, solution would be to make all implicit changes to dynamic intervals explicit by having the differential nodes generate differentials, at the start of each chronon, for tuples containing dynamic intervals in their attribute time-stamps. A more practical solution would be to identify at each node, for tuples containing a dynamic interval, the future time, if any, when the interval would cause the node's output to change. These tuples could then be queued at the node for reprocessing at that time. If a differential changing that tuple arrived in the interim, the queued tuple would simply be dequeued. Union, difference, projection, and historical derivation nodes all would have to be augmented to perform this task. Selection and cartesian product nodes, because they do not deal with attribute time-stamps, would be unaffected.

7.5.7 Deferred View Materialization

The TQuel prototype supports only immediate-incremental view materialization. Our architecture, however, also accommodates deferred-incremental view materialization. Under deferred view materialization, a view is not updated immediately after each change to one of its underlying relations but only just before each access to the view itself. Deferred view materialization has several advantages over immediate view materialization [Horwitz 1985, Roussopoulos & Kang 1986A, Roussopoulos 1987]. First, the completion of transactions that change base relations are not delayed while views are being updated. Second, differentials can be collected and consolidated to reduce traffic through the network and processing costs at nodes. Finally, views may be updated as a background task to make use of otherwise unused resources. The obvious disadvantage of deferred view materialization is that access to a view may be delayed while the view is being brought up-to-date, although users can eliminate this delay by accessing an "almost up-to-date" copy of the view [Roussopoulos 1987]. Also, deferred view materialization is not appropriate for all applications. For example, it would be inappropriate if views were being used to drive real-time display systems.

To implement deferred rather than immediate view materialization, differential nodes collect and consolidate differentials, but only propagate the consolidated differentials to their children on demand. Just before a view is to be accessed, the differential nodes for the base relations upon which the view depends propagate their differentials to the view's update node. Only after the propagated differentials have been processed by the view's update node, may the view be accessed. Although simple in concept, deferred view materialization requires additional data structures and control mechanisms not needed for immediate view materialization. For example, data structures are needed to store differentials at differential nodes and control mechanisms are needed to record whether views are up-to-date and to initiate the updating of views as needed.

Also, node sharing in update networks further complicates deferred view materializa-

tion. For example, a base relation may be an underlying relation in multiple subnetworks associated with a single view; it also may be an underlying relation of an arbitrary number of views. Hence, a differential node may have multiple children associated with the same view as well as children associated with different views. When a differential node is instructed to propagate its differential to a view, does it propagate the differential to all its children or only to children of that view? If it propagates the differential to all its children, access to the view that needs to be updated may be delayed arbitrarily long, and resources may be wasted updating views with differentials that may be negated before the views are accessed. If, however, the node propagates the differential only to its children for one view, each edge that emanates from the node must be associated with a specific view or set of views (if view update networks are integrated using optimization techniques) and data structures for storing differentials must be made more complex to record which differentials have been propagated to which children and to indicate when differentials have been propagated to all children and can be removed. Roussopoulos describes one technique for efficient maintenance of this information [Roussopoulos & Kang 1986A]. In this technique time-stamp records are inserted into a node's differential file to record the number of children to whom the preceding portion of the differential has been propagated. Only when this count equals the node's number of children, can the preceding portion of the file be removed. Also, the time-stamp for the lastest update to each view must be recorded between updates to indicate the portion of differentials that have already been applied to the view. Note also that, if node sharing among view update networks is allowed at operator nodes, they too must deal with these same implementation issues.

7.5.8 Concurrency Control and Recovery

We now consider the applicability of existing techniques for concurrency control and recovery to temporal databases in which update networks are used to maintain materialized views incrementally. Figure 7.9 shows how a standard model of concurrency control and recovery in conventional, non-temporal DBMS's [Bernstein et al. 1987] could be adapted for use in our TQuel prototype. Here the semantic analyzer, code generator, and interpreter all issue read, write, commit, and abort operations to the *transaction manager*, which performs any necessary preprocessing before forwarding the operations to the *scheduler*. The *scheduler*, which is responsible for the concurrent execution of the active transactions, then orders the operations so that their execution will be both serializable and recoverable. The *recovery manager* processes read, write, commit, and abort operations issued by the scheduler atomically to ensure that their execution is serializable. The *cache manager* moves data between stable storage and volatile storage using its *fetch* and *flush* operations. The *recovery manager* partially controls the *cache manager*'s flush operations to ensure that operations, once executed, are recoverable [Bernstein et al. 1987].

As in a non-temporal DBMS, base relations, along with the system catalog, reside in stable storage and are recoverable following a failure. The database update network and the intermediate relation states cached at the nodes in the network may, but need not, reside in stable storage. The update network and its intermediate relation states

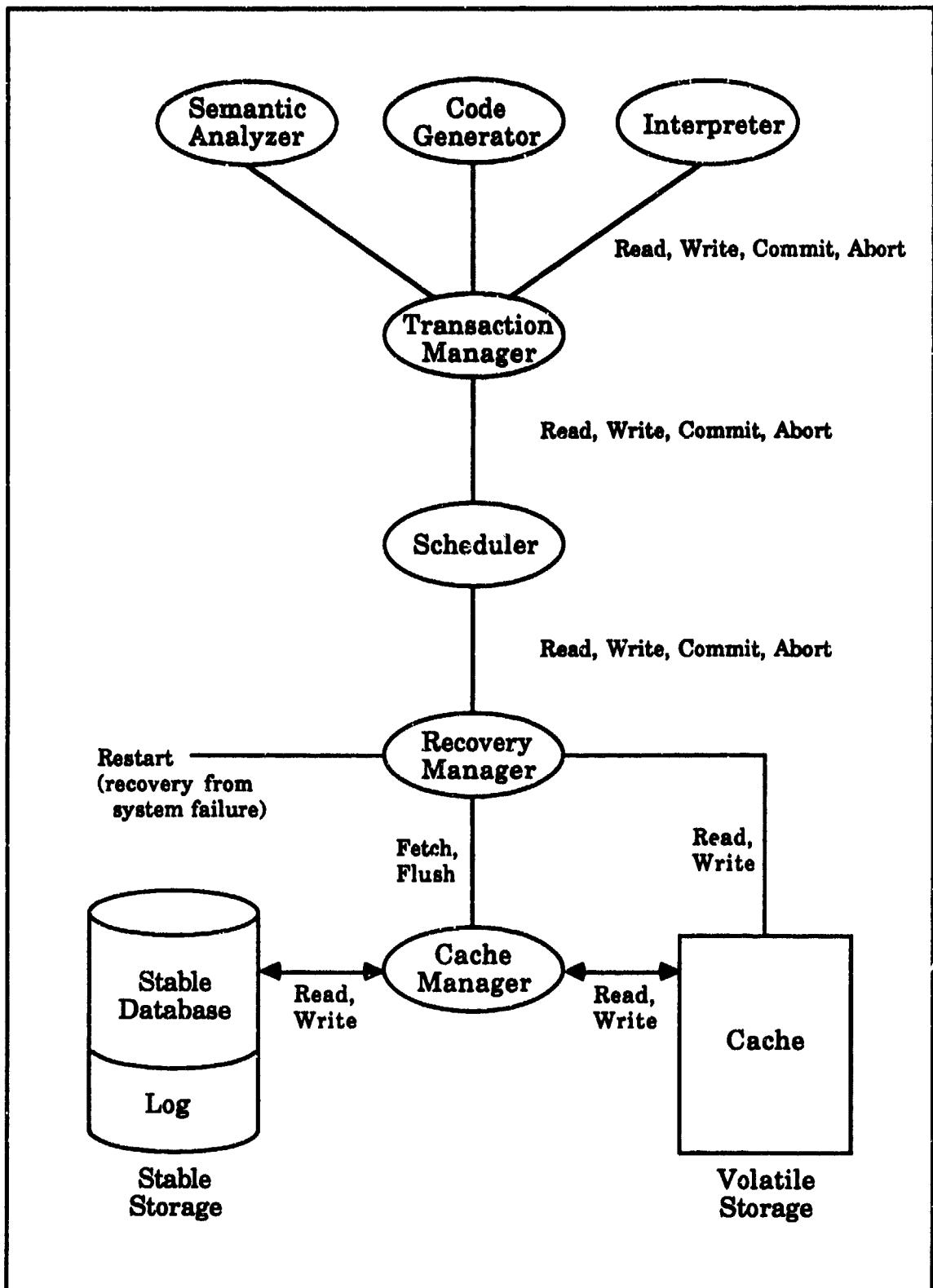


Figure 7.9: Model of Concurrency Control and Recovery [Bernstein et al. 1987]

can always be reconstructed from the base relations and view definitions in stable storage following a system failure. Hence, the update network and its intermediate relation states, unlike base relations and view definitions, need not be recoverable. The update network (or some portion of the network) may, however, be stored in stable storage to eliminate the need to reconstruct the network following a failure. Whether the update network, or some portion of the network, is made recoverable, is an efficiency issue that depends on the cost of maintaining a recoverable network, the cost of reconstruction, and the probability of failure.

Concurrency Control

Although our TQuel prototype supports only sequential processing of differentials through the database update network, standard locking mechanisms [Bernstein et al. 1987] can be used to allow concurrent retrieval and update of views in the network, whether an immediate or deferred materialization strategy is used. If the immediate view materialization strategy is used, conservative two-phase locking can be used to lock for update all the base relations to be updated, along with the nodes in their view dependency graphs, at the start of each transaction. Then, strict two-phase locking can be used to release these locks only after the transaction is committed.

Other more optimistic locking strategies also can be used, at the expense of possible cascading aborts if deadlock occurs or a transaction is aborted. For example, locking of a base relation, along with its view dependency graph, may be delayed until just before it is updated. Similarly, once the base relation is updated and the differential computed, the locks on the nodes in the base relation's view dependency graph may be released as the differential for the update is propagated through the network. If depth-first search is used to propagate the differential, a node's lock may be released as the propagation is completed at the subtree rooted at that node. Alternatively, if breadth-first search is used, a node's lock may be released as the propagation is completed at that node. Another optimistic locking protocol delays the locking of a node in a base relation's view dependency graph until just before the node is to process a differential and then releases the lock immediately after the node has processed the differential. In this locking protocol, a transaction that accesses a view must lock for retrieval all base relations used to derive the view while the view is being accessed. This latter action is necessary to ensure that a view is consistent with its underlying relations when it is accessed.

If the deferred view materialization strategy is used, a variation of two-phase locking proposed by Roussopoulos can be used [Roussopoulos 1987]. When a view is to be updated, all base relations that are used to derive the view are locked for retrieval to prevent their update during the update of the view, and all other nodes in the view's update network are locked for update. Locks at the nodes are then released as the differentials are propagated toward the view's update node.

Although a locking protocol is needed to control concurrent access to snapshot and historical relations and to the current states of rollback and temporal relations, the protocol

need not be extended to control concurrent access to the past states of rollback and temporal relations. Past states of rollback and temporal relations are read-only; once recorded, they can't be changed. Hence, a rollback operation, because it accesses a past state of a rollback or a temporal relation, need never be delayed, even if the relation it is accessing is locked for update.

Concurrent execution of transactions may complicate slightly the implementation of rollback operations. Rollback operators roll a relation back to its state at a specified time (i.e., the state of the relation immediately following the most recently committed transaction on the relation before the specified time). Hence, to support rollback operations, a transaction's commit time needs to be assigned to each change the transaction makes to the database. Because a transaction's commit time is not known until the end of the transaction, however, it can't be recorded when changes are made. Rather than update the database after the transaction is committed to record the transaction's commit time with each change, it may be more efficient to record a transaction number (e.g., transaction start time) with each change and to maintain a mapping from transaction commit times onto transaction numbers. If transactions are processed sequentially, the elements in a relation's class, signature, and state sequences will be ordered by transaction number as well as transaction commit time. If, however, transactions are processed concurrently, the elements in each of a relation's three sequences, while ordered by transaction commit time, may not necessarily be ordered by transaction number. Rolling back a relation to a specified time is slightly more complicated in the latter case because the transaction numbers assigned to elements in each of the relation's three sequences do not necessarily form a linearly ordered search space. Hence, rolling back a relation to a specified time also may require multiple accesses to a mapping from transaction numbers onto transaction commit times. A time-stamp concurrency control protocol [Bernstein et al. 1987], however, could be used to support the concurrent execution of transactions while ensuring that each relation's class, signature, and state sequence was ordered by transaction number as well as commit time.

Recovery

Although recovery of base relations and the system catalog is necessary, recovery of the update network, materialized views, and the intermediate relation states cached at operator nodes is not. Any portion of the update network can always be reconstructed from the base relations and the system catalog, if necessary. Recovery of the update network may, however, be desirable for efficiency of restart following a system or device failure. In this case, standard techniques for recovery management of base relations [Bernstein et al. 1987] can be used for recovery management of intermediate relation states, assuming they are cached in materialized form. Also, recovery capabilities need not extend to the entire network. Any subnetwork rooted at base relations may be recoverable without the entire network being recoverable. Finally, differentials can be used to recover the update network from a transaction abort. A reverse differential (i.e., before images and after images are switched) is simply propagated through a base relation's view dependency graph to negate

the effects of the original differential.

7.5.9 Aggregates

Incremental computation of aggregates presents the same problem in temporal-database update networks as in snapshot-database update networks. Some aggregates, such as `sum` and `avg`, can be updated incrementally without problem, while other aggregates, such as `min` and `max`, may require that their values be recomputed on a change to their underlying relations (e.g., a tuple containing the value of a `max` aggregate is deleted). To improve the efficiency of maintaining aggregates of this latter type, Hanson suggests that a queue of possibly duplicate candidate aggregate values, rather than a single value, be maintained [Hanson 1987B]. Then, if a change to an aggregate's underlying relation changes or deletes a tuple containing the aggregate's value, the aggregate's new value could be assumed to be the next element in the queue. Only if the queue were empty, would the aggregate have to be recomputed. This technique can be extended to apply to historical aggregates by maintaining queues for each chronon at which the aggregate has a value. Appropriate data structures for maintaining such queues have yet to be studied.

7.6 Summary

In this chapter we discussed an architecture for query processing in TDBMS's that accommodates the incremental maintenance of materialized views. We then described a prototype query processor for TQuel that we built using this architecture. Construction of this prototype is an existence proof that the paradigm for incremental expression evaluation, along with the incremental snapshot and historical algebras defined in the previous chapter, is adequate to implement incremental view materialization in TDBMS's.

We also identified problems that arise when materialized historical views are maintained incrementally and proposed various techniques for resolving those problems. Our historical algebra is, in many respects, similar to the snapshot algebra. Hence, similar problems arise when either materialized snapshot or materialized historical views are maintained incrementally. Also, the solutions to the problems are often similar, if not the same, for both snapshot and historical views. For example, the techniques for global and local optimization of update networks, cacheing of intermediate relation states, and concurrency control and recovery apply equally to update networks for either snapshot or historical views. Also, although particular to historical views, representation of attribute time-stamps and historical differentials is straightforward. Other problems, however, either have no analogue in snapshot databases or are not amenable to simple solution. These include the problems of accommodating dynamic time-stamps efficiently, reducing the search space of interval assignments at historical derivation nodes, and implementing all aggregates efficiently. Additional research will be required to find solutions to these problems.

In the next chapter, we review other proposals for adding valid and transaction time to the snapshot algebra, identify a set of properties desirable of such extensions, and compare

our approach and those of others, using the properties as evaluation criteria.

Chapter 8

Evaluation Criteria

In Chapters 3 and 4 we identified several basic design decisions that one must make to extend the snapshot algebra to handle valid time and transaction time.

- Is valid time associated with tuples or with attributes?
- How is valid time represented? Are time-stamps, which represent valid time, chronons, intervals, or sets of chronons, not all of which are consecutive?
- Are attributes required to be atomic-valued or are they allowed to be set-valued?
- Is the set-theoretic semantics of the basic relational operators retained and new operators introduced to deal with the temporal dimension of the real-world phenomena being modeled or is the semantics of the relational operators extended to account for the temporal dimension directly? If the semantics of the relational operators is extended to handle time, how do these operators compute the valid time of the attributes in resulting tuples?
- How does the algebra handle time-oriented operations like temporal selection, temporal projection, and temporal aggregation?
- Is transaction time associated with attributes, tuples, or relation states?

Although the choices one makes for these design decisions determine important properties of the resulting algebraic language, we stated our choices in Chapters 3 and 4 without explanation. In this chapter we motivate our choices.

Over the past decade, no less than 11 temporal extensions of the snapshot algebra (including ours) have been proposed, some with several variants. Most of these proposals support only valid time and can be termed *historical algebras*. Others, like ours, support both valid time and transaction time. We hereafter refer to these as *temporal algebras*. Even with this significant interest in temporal extensions of the snapshot algebra, previous research has not focused on the properties that historical and temporal algebras should

have. A set of well-defined, objective criteria for judging the relative merit of these various algebras has yet to be proposed. Hence, we identify here a set of 29 criteria for evaluating temporal extensions of the snapshot algebra. These criteria, although not all compatible, are well-defined, have an objective basis for being evaluated, and are arguably beneficial.

Two important benefits accrue from the identification of a comprehensive set of evaluation criteria. First, the criteria provide a means for objective evaluation of algebras in terms of their properties. Second, the criteria can be used as a guide in making design decisions that will result in an algebra with a maximal subset of desirable properties.

After a brief review of the algebras proposed by others, we present our set of 29 criteria for evaluating temporal extensions of the snapshot algebra. We then evaluate our algebra and those proposed by others against the criteria. We conclude the chapter with a review of our design decisions. We explain how our goal to define an algebra with as many desirable properties as possible led us to choose the design options we did.

8.1 Temporal Extensions of the Snapshot Algebra

In this section we review briefly 10 temporal extensions of the snapshot algebra. We describe the extensions in terms of the types of objects that each defines and the operations on object instances that each provides. We also emphasize the choices made for each of the key design decisions. All these extensions support valid time. Only one, Ben-Zvi's Time Relational Model [Ben-Zvi 1982], supports both valid time and transaction time.

LEGOL 2.0 [Jones et al. 1979] is a language based on the relational model designed to be used in database applications, such as legislative rules writing and high-level system specification, in which the temporal ordering of events and the valid times for objects are important. Objects in the LEGOL 2.0 data model are relation states as in the relational data model, with one distinction. Tuples in LEGOL 2.0 are assigned two implicit time attributes, *start* and *stop*. The values of these two attributes are the chronons corresponding to the end-points of the interval of existence (i.e., valid time) of the real-world object or relationship represented by a tuple.

EXAMPLE. Examples in this section show the semantically equivalent representation of historical state S_1 from page 25 of Chapter 3 in the algebras reviewed. As in Chapter 3, S_1 is a historical state over the signature *Student* with explicit attributes {*sname*, *course*}. The granularity of time continues to be a semester relative to the Fall semester 1980. Because the algebras all define relation states differently and, in some cases, require implicit attributes, we show all examples of relation states in this chapter in tabular form for both clarity and consistence of notation. Here, S_1 is a historical state in LEGOL 2.0.

$S_1 =$

| sname | course | start | stop |
|----------|-----------|-------|------|
| "Phil" | "English" | 1 | 1 |
| "Phil" | "English" | 3 | 4 |
| "Norman" | "English" | 1 | 2 |
| "Norman" | "Math" | 5 | 6 |

Note that two value-equivalent tuples are needed to record Phil's enrollment in English, as his enrollment was not continuous. \square

Operations in LEGOL 2.0 are not defined formally, although the more important operations are described using examples. LEGOL 2.0 retains the standard set-theoretic operations and introduces several time-related operations to handle the temporal dimension of data. The new time-related operations are time intersection, one-sided time intersection, time union, time difference, and time-set membership. Time intersection acts as a temporal join, where the valid time of each output tuple is computed using *intersection semantics* (i.e., the valid time of each output tuple is the intersection of the valid times of two overlapping input tuples). Although the semantics of the other time-related operations is left unspecified, these operators appear to support a limited form of temporal selection as well as a temporal join using *union semantics* (i.e., the valid time of each output tuple is the union of the valid times of two overlapping input tuples).

The Time Relational Model [Ben-Zvi 1982] supports both valid time and transaction time. Two types of objects are defined: snapshot relation states, as defined in the snapshot algebra, and temporal relation states. Temporal relation states are set of tuples, with each tuple having five implicit time attributes. The attributes **effective-time-start** and **effective-time-stop** are the end-points of the interval of existence of the real-world phenomenon being modeled, **registration-time-start** and **registration-time-stop** are the end-points of the interval when the tuple is logically a tuple in the relation state, and **deletion-time** records the time when erroneously entered tuples are logically deleted.

EXAMPLE. S_1 is a temporal relation state in the Time Relational Model on the relation signature *Student* with explicit attributes {sname, course}. For completeness, we assume that the tuples' effective start times were recorded by the transaction corresponding to transaction number 423 and their effective stop times were recorded by the transaction corresponding to transaction number 487. We also assume that none of the tuples has yet to be deleted.

| $s_1 =$ | ename | course | effective time-start | effective time-stop | registration time-start | registration time-step | deletion time |
|---------|--------------|---------------|-----------------------------|----------------------------|--------------------------------|-------------------------------|----------------------|
| | "Phil" | "English" | 1 | 1 | 423 | 487 | — |
| | "Phil" | "English" | 3 | 4 | 423 | 487 | — |
| | "Norman" | "English" | 1 | 2 | 423 | 487 | — |
| | "Norman" | "Math" | 5 | 6 | 423 | 487 | — |

□

A new *Time-View* operator, $TV = (t_e, t_s)$, is introduced that maps a temporal relation state onto a snapshot state. The Time-View operator can be thought of as a limited form of temporal selection that selects from the relation's state at transaction time t_s those tuples with a valid time of t_e . Once the specified tuples are selected, however, the Time-View operator discards their implicit time attributes to construct a snapshot state.

EXAMPLE. If we let $TV = (1, 423)$, then

| $TV(s_1) =$ | ename | course |
|-------------|--------------|---------------|
| | "Phil" | "English" |
| | "Norman" | "English" |

□

The semantics of the five relational operators union, difference, join, selection, and projection is extended to handle both the valid time and the transaction time of tuples directly. These operators, like the Time-View operator, are all defined in terms of a transaction time t_s and a valid time t_e . Input tuples are restricted to those tuples in an input relation's state at transaction time t_s having a valid time of t_e ; the valid times of all tuples that participate in an operation are thus guaranteed to overlap at time t_e . Each operator computes the valid time of its output tuples from the valid times of qualifying tuples in its input relation states using either union or intersection semantics. For example, the union operator is defined using union semantics and the join operator is defined using intersection semantics. The valid time of tuples resulting from the difference operator, however, is left unspecified.

The Temporal Relational Model [Navathe & Ahmed 1986] allows both non-time-varying and time-varying attributes, but all of a relation's attributes must be the same type. Objects are snapshot relation states, whose attributes are all non-time-varying, and historical relation states, whose attributes are all time-varying. The end-points of the interval of validity of tuples in historical states are recorded in two mandatory time attributes, **time-start** and **time-end**. Hence, the structure of a historical state in the Temporal Relational Model is the same as that of a historical state in LEGOL 2.0, as shown on page 208. Value-equivalent tuples, although allowed, are required to be coalesced. The set theoretic operators are retained and five additional operators on time-varying relation

states are introduced. The operators *Time-Slice*, *Inner Time-View*, and *Outer Time-View* are all forms of temporal selection. *TCJOIN* and *TCNJOIN* are both join operators defined using intersection semantics. Two other join operators, *TJOIN* and *TNJOIN*, are discussed. They retain the time-stamps of underlying tuples in their resulting tuples but are, therefore, outside the algebra (the domain of the operators contains objects not defined by the model).

In Sadeghi's algebra [Sadeghi 1987], objects are historical relation states. Two implicit attributes, *start* and *stop*, record the end-points of each tuple's interval of validity. Hence, the structure of a historical state in Sadeghi's algebra is also the same as that of the historical state in LEGOL 2.0, as shown on page 208. Sadeghi's algebra, like Navathe's Temporal Relational Model, allows value-equivalent tuples and requires that value-equivalent tuples be coalesced. Historical versions of the snapshot operators union, difference, cartesian product, selection, projection, and join are defined. Both cartesian product and join are defined using intersection semantics. A new operator, *WHEN*, is introduced. It maps a historical relation state onto the intervals that are the time-stamps of tuples in the relation state. Whether the result of this operation is another type of object or a historical state without explicit attributes is unclear.

Sarda's algebra [Sarda 1988] is another historical algebra that associates valid time with tuples. Objects can be either snapshot or historical relation states. Unlike the algebras mentioned previously, Sarda's algebra represents valid time in a historical relation as a single, non-atomic, implicit attribute named *period*.

EXAMPLE. S_1 is a historical relation state in Sarda's algebra on the relation signature *Student* with explicit attributes {*sname*, *course*}.

| $S_1 =$ | <i>sname</i> | <i>course</i> | <i>period</i> |
|---------|--------------|---------------|---------------|
| | "Phil" | "English" | 1...2 |
| | "Phil" | "English" | 3...5 |
| | "Norman" | "English" | 1...3 |
| | "Norman" | "Math" | 5...7 |

Also unlike the other algebras, a tuple in Sarda's algebra isn't considered valid at its right-most boundary point. Hence, the first tuple signifies that Phil was enrolled in English during the Fall semester 1980, but not during the Spring semester 1981. \square

Sarda's algebra retains the basic semantics of some of the set theoretic operators, extends the definition of one operator to handle valid time directly, and introduces several new operators. Projection and cartesian product are defined to treat the implicit attribute *period* the same as they would an explicit attribute. Projection maps a historical state onto either a snapshot or a historical state, depending on whether the implicit attribute *period* is a projection attribute. Similarly, cartesian product simply combines tuples from two historical states, without discarding or changing their time-stamps. Hence, the result of a cartesian product isn't a historical state but a snapshot state with two non-

atomic attributes. The semantics of the select operator, however, is extended to allow for both temporal, as well as, non-temporal predicates. Whether the algebra retains the set theoretic semantics of union and difference is left unspecified. *EXPAND*, *CONTRACT*, *PROJECT-AND-WIDEN*, and *CONCURRENT PRODUCT* are the new operators. *EXPAND* produces, for each chronon in the time-stamp of each tuple in a historical state, a value-equivalent tuple with that chronon as its time-stamp. *CONTRACT*, the inverse of *EXPAND*, coalesces value-equivalent tuples. *PROJECT-AND-WIDEN* is a form of temporal projection that coalesces value-equivalent tuples and *CONCURRENT PRODUCT* is cartesian product defined using intersection semantics.

Unlike the algebras discussed above, the Temporal Relational Algebra [Lorentzos & Johnson 1987A] associates time-stamps with individual attributes rather than with tuples. Although a time-stamp is normally associated with all the attributes in a tuple, a time-stamp may be associated with any non-empty subset of attributes in a tuple. Furthermore, no implicit or mandatory time-stamp attributes are assumed. Time-stamps are simply explicit, numeric-valued attributes. They represent either the chronon during which one or more attribute values are valid or a *boundary point* of the interval of validity for one or more attribute values. Several time-stamp attributes may also be used together to represent a chronon of nested granularity.

EXAMPLES. First, let S_1 be a historical relation state in the Temporal Relational Algebra on the relation signature *Student* with attributes {*sname*, *n-start*, *n-stop*, *course*, *c-start*, *c-stop*}. Unlike the other algebras, the time-stamp attributes appear as explicit attributes in the relation signature. Here we assume that the attributes *n-start* and *n-stop* represent the boundary points of the interval of validity for the attribute *sname* and the attributes *c-start* and *c-stop* represent the boundary points of the interval of validity for the attribute *course*. Note, however, that we could have specified the same time-stamp attributes for both *sname* and *course* in this example.

| $S_1 =$ | <i>sname</i> | <i>n-start</i> | <i>n-stop</i> | <i>course</i> | <i>c-start</i> | <i>c-stop</i> |
|---------|--------------|----------------|---------------|---------------|----------------|---------------|
| | "Phil" | 1 | 2 | "English" | 1 | 2 |
| | "Phil" | 3 | 5 | "English" | 3 | 5 |
| | "Norman" | 1 | 3 | "English" | 1 | 3 |
| | "Norman" | 5 | 7 | "Math" | 5 | 7 |

A time-stamp in the Temporal Relational Algebra, like one in Sarda's algebra, doesn't include its right-most boundary point.

Now let R_1 be a historical relation state in the Temporal Relational Algebra on the relation signature *Student* with attributes {*sname*, *course*, *semester-start*, *semester-stop*, *week-start*, *week-stop*}, where all four time-stamp attributes are associated with both *sname* and *course*. Assume that the granularity for the time-stamp attributes *week-start* and *week-stop* is a week relative to the first week of a semester.

| R ₁ = | sname | course | semester-start | semester-stop | week-start | week-stop |
|------------------|-----------|--------|----------------|---------------|------------|-----------|
| "Phil" | "English" | | 1 | 2 | 1 | 9 |
| "Phil" | "English" | | 3 | 5 | 1 | 17 |
| "Norman" | "English" | | 1 | 3 | 1 | 9 |
| "Norman" | "Math" | | 5 | 7 | 9 | 17 |

In this example, we specify the weeks during a semester when a student was enrolled in a course. For example, Phil was enrolled in English during the Fall semester 1980 for only the first 8 weeks of the semester. Note that the meaning of the week-start and week-stop attributes is relative to the semester-start and semester-stop attributes. □

The standard set-theoretic operations are retained in the Temporal Relational Algebra unchanged. Although no new time-oriented operations are introduced, three new operators, *EXTEND*, *UNFOLD*, and *FOLD*, which are defined in terms of the conventional relational operators, are introduced. These operators allow conversion between relation states whose tuples contain two time-stamp attributes representing the end-points of the interval of validity of one or more attributes to equivalent relation states whose tuples contain a single time-stamp attribute representing a chronon during which the same attributes are valid. Relation states whose tuples contain only time-stamp attributes representing the end-points of intervals of validity are considered to be *folded* while relation states whose tuples contain only time-stamp attributes representing individual chronons of validity are considered to be *unfolded*. Relation states S₁ and R₁ in the above examples are folded.

EXAMPLE. Let R₂ be an equivalent representation of R₁ in which the two time-stamp attributes semester-start and semester-stop have been unfolded onto a single time-stamp attribute semester.

| R ₂ = | sname | course | semester | week-start | week-stop |
|------------------|-----------|--------|----------|------------|-----------|
| "Phil" | "English" | | 1 | 1 | 9 |
| "Phil" | "English" | | 3 | 1 | 17 |
| "Phil" | "English" | | 4 | 1 | 17 |
| "Norman" | "English" | | 1 | 1 | 9 |
| "Norman" | "English" | | 2 | 1 | 9 |
| "Norman" | "Math" | | 5 | 9 | 17 |
| "Norman" | "Math" | | 6 | 9 | 17 |

We could now apply *UNFOLD* once more to unfold the attribute week-start and the attribute week-stop onto a single time-stamp attribute week. The resulting relation would have 72 tuples. □

The Historical Relational Data Model [Clifford & Croker 1987] allows two types of objects: a set of chronons, termed a *lifespan*, and a historical relation state, where each

attribute in the relation scheme and each tuple in the relation state is assigned a lifespan. A relation scheme in the Historical Relational Data Model is an ordered four-tuple containing a set of attributes, a set of key attributes, a function that maps attributes to their lifespans, and a function that maps attributes to their value domains. A tuple is an ordered pair containing the tuple's value and its lifespan. Attributes are not atomic-valued; rather, an attribute's value in a given tuple is a partial function from the domain of chronons onto the attribute's value domain, defined for the attribute's valid time (i.e., the intersection of the attribute and tuple lifespans). Relations have key attributes and no two tuples in a relation state are allowed to match on the values of the key attributes at the same chronon.

EXAMPLE. S_1 is a historical relation in the Historical Relational Data Model on the relation signature *Student*, where $\{\text{sname} \rightarrow \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}, \text{course} \rightarrow \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}\}$ is the function assigning lifespans to attributes.

| $S_1 =$ | Tuple Value | | Tuple Lifespan |
|--------------|---------------|--------|----------------|
| | sname | course | |
| 1 → "Phil" | 1 → "English" | | {1, 3, 4} |
| | 3 → "English" | | |
| | 4 → "English" | | |
| 1 → "Norman" | 1 → "English" | | {1, 2, 5, 6} |
| | 2 → "English" | | |
| | 5 → "Math" | | |
| | 6 → "Math" | | |

Because tuple lifespans are sets and because both Phil and Norman were never enrolled in more than one course at the same time, we are able to record each of their enrollment histories in a single tuple. If one had been enrolled in two or more courses at the same time, however, his total enrollment history could not have been recorded in a single tuple as attribute values are functions from a lifespan onto a value domain. Note also that we have chosen the most straightforward representation for an attribute whose value is a function. Because attribute values in both Clifford's algebra and Gadia's algebras, which we describe next, are functions, they have an arbitrary number of other physical representations. □

The semantics of the relational operators union, difference, intersection, projection, and cartesian product is extended to handle lifespans directly. For example, the lifespan of each tuple output by cartesian product is the union of the lifespans of the two tuples in the input relation states that contribute to the output tuple. A null value is assigned to an attribute in the output tuple for each chronon that is in the lifespan of the output tuple but not in the lifespan of the input tuple associated with that attribute. Also, temporal versions of Θ -join, equi-join, and natural join are defined using intersection semantics and several new time-oriented operations are introduced. *WHEN* maps a relation state onto its lifespan, where the lifespan of a relation state is defined to be the union of the lifespans of its tuples (e.g., $\{1, 2, 3, 4, 5, 6\}$ in the above example). *SELECT-IF* is a form of temporal

selection that selects tuples that are both valid and satisfy a given selection criterion at a specified time and *TIME-SLICE* is a form of temporal projection that restricts the tuple lifespans of its resulting tuples to some portion of their original lifespans. The operator *SELECT-WHEN* possesses features of both temporal selection and temporal projection; it is a variant of *SELECT-IF* that restricts the tuple lifespans of its resulting tuples to the times when they satisfy the selection condition. Finally, a *TIME-JOIN* operator is defined that restricts the tuple lifespans of its resulting tuples to the value of a time-valued attribute.

Gadia's homogeneous model [Gadia 1988] also allows two types of objects: *temporal elements* and historical relation states. A temporal element is a finite union of disjoint intervals (effectively a set of chronons) and attribute values are functions from temporal elements onto attribute value domains. The model requires that all attribute values in a given tuple be functions on the same temporal element. This property, termed *homogeneity*, ensures that a snapshot of a historical relation state at time t always produces a conventional snapshot state without nulls.

EXAMPLE. S_1 is a historical relation state in Gadia's homogeneous model over the signature *Student* with attributes {*sname*, *course*}.

| $S_1 =$ | <i>sname</i> | <i>course</i> |
|---------|--|---|
| | $[1, 2) \cup [3, 5) \rightarrow \text{"Phil"}$ | $[1, 2) \cup [3, 5) \rightarrow \text{"English"}$ |
| | $[1, 3) \cup [5, 7) \rightarrow \text{"Norman"}$ | $[1, 3) \rightarrow \text{"English"}$ $[5, 7) \rightarrow \text{"Math"}$ |

Here the interval $[t_1, t_2)$ is the set of chronons $\{t_1, \dots, t_2 - 1\}$. Again, we are able to record the enrollment histories of Phil and Norman in single tuples only because they were never enrolled in more than one course at the same time. \square

A historical version of each of the five basic conventional relational operators is defined using snapshot semantics. For each historical operator, the snapshot of its resulting historical relation state at time t is required to equal the result obtained by applying the historical operator's relational counterpart to the snapshot of the underlying historical relation states at time t . Two new operators are also introduced. One, *tdom*, maps either a tuple or a relation state onto its temporal domain, where the temporal domain of a tuple is its temporal element and the temporal domain of a relation state is the union of its tuples' temporal elements. For example, the temporal domain of S_1 above is $[1, 7)$. The other operator, termed *temporal selection*, is a limited form of both temporal selection and temporal projection; it selects from a relation state those tuples whose temporal elements overlap a specified temporal element and restricts attribute values in the resulting tuples to the intersection of their temporal elements and the specified temporal element.

Gadia's multihomogeneous model [Gadia 1986] and Gadia's and Yeung's heterogeneous models [Gadia & Yeung 1988, Yeung 1986] are all extensions of the homogeneous

model. They lift the restriction that all attribute values in a tuple be functions on the same temporal element. We consider here only the latest [Gadia & Yeung 1988] of these extensions. Temporal elements may be multi-dimensional to model different aspects of time (e.g., valid time and transaction time). Attribute values are still functions from temporal elements onto attribute value domains, but attribute values need not be functions on the same temporal element. Relations are assumed to have key attributes, with the restriction that the range of the function assigned to each key attribute in a tuple be a single element of the attribute's value domain. Also, no two tuples may match on the ranges of the functions assigned to the key attributes. Hence, in the previous example, the attribute *sname* would qualify as a key attribute in the heterogeneous model. The semantics of union, cartesian product, selection, projection, and join are extended to account for temporally heterogeneous attribute values. Also, temporal variants of selection and join are introduced. The semantics of difference and intersection, however, are left unspecified.

Tansel's historical algebra [Tansel 1986] allows only one type of object: the historical relation state. However, four types of attributes are supported, the attributes of a relation need not be the same type, and attribute values in a given tuple need not be homogeneous. Attributes may be either non-time-varying or time-varying and they may be either atomic-valued or set-valued. The value of a time-varying, atomic-valued attribute is represented as a triplet containing an element from the attribute's value domain and the boundary points of its interval of existence while the value of a time-varying, set-valued attribute is simply a set of such triplets.

EXAMPLE. S_1 is a historical relation state in Tansel's algebra over the relation signature *Student* with attributes {*sname*, *course*}, where *sname* is a non-time-varying, atomic-valued attribute and *course* is a time-varying, set-valued attribute.

| $S_1 =$ | <i>sname</i> | <i>course</i> |
|---------|--------------|---|
| | "Phil" | { ([1, 2), "English"), ([3, 5), "English") } |
| | "Norman" | { ([1, 3), "English"), ([5, 7), "Math") } |

Because Tansel doesn't define time-varying attributes as functions, the enrollment history of a student can be recorded in a single tuple, even if the student was enrolled in two or more courses at some time. Note, however, that each interval of enrollment, even for the same course, must be recorded as a separate element of a time-varying, set-valued attribute. \square

The conventional relational operators are extended to account for the temporal dimension of data and several new time-related operations are introduced. *PACK* combines tuples whose attribute values differ for a specified attribute but are otherwise equal. Conversely, *UNPACK* replicates a tuple for each element in one of its set-valued attributes.

T-DEC decomposes a time-varying, atomic-valued attribute in a historical relation state into three non-time-varying, atomic-valued attributes, representing the three components of the time-varying, atomic-valued attribute. Conversely, *T-FORM* combines three non-time-varying, atomic-valued attributes, representing a value and the boundary points of the value's interval of validity into a single time-varying, atomic-valued attribute. *DROP-TIME* discards the time components of a time-varying attribute. Finally, *SLICE*, *USLICE*, and *DSLICE*, are limited forms of temporal projection in which the time-stamp of a time-varying attribute is recomputed as the intersection, union, and difference, respectively, of its original time-stamp and the time-stamp of another specified attribute. If the recomputed time-stamp is empty, the tuple is discarded. Tansel also introduces a new operation, termed *enumeration*, to support aggregation [Tansel 1987]. The enumeration operator derives, for a set of chronons or intervals and a historical state, a table of data to which aggregate operators (e.g., count, avg, min) can be applied.

EXAMPLES. Let R_1 be the historical relation state, resulting from the unpacking of attribute *course* of S_1 in the previous example, over the relation signature *Student* with attributes {*sname*, *course*}, where *sname* is a non-time-varying, atomic-valued attribute and *course* is a time-varying, atomic-valued attribute.

| $R_1 =$ | <i>sname</i> | <i>course</i> |
|---------|--------------|-----------------------|
| | "Phil" | ([1, 2), "English") |
| | "Phil" | ([3, 5), "English") |
| | "Norman" | ([1, 3), "English") |
| | "Norman" | ([5, 7), "Math") |

Now, let R_2 be the historical relation state, resulting from the decomposition (*T-DEC*) of attribute *course* of relation R_1 , over the relation signature *Student* with attributes {*sname*, *course*, *course_L*, *course_U*}, where *sname*, *course*, *course_L*, and *course_U* are all non-time-varying, atomic-valued attributes.

| $R_2 =$ | <i>sname</i> | <i>course</i> | <i>course_L</i> | <i>course_U</i> |
|---------|--------------|---------------|---------------------------|---------------------------|
| | "Phil" | "English" | 1 | 2 |
| | "Phil" | "English" | 3 | 5 |
| | "Norman" | "English" | 1 | 3 |
| | "Norman" | "Math" | 5 | 7 |

□

Table 8.1 and Table 8.2 are a summary of the features of the 10 algebras described above and the algebra defined in the previous chapters of this dissertation. These tables show the range of solutions chosen by the developers of the algebras to the first five design

| TIME-STAMP REPRESENTATION | | | |
|----------------------------|--|----------------------------|---|
| | single chronon | interval (two chronons) | set of chronons |
| Time-stamped Tuples | Jones Ben-Zvi Navathe & Ahmed Sadeghi | Sarda | Clifford & Croker |
| | Lorentzos & Johnson | Tansel | Clifford & Croker Gadia Gadia & Yeung McKenzie |
| Time-stamped Attributes | | | |
| | | | |

Table 8.1: Representation of Time in the Algebras

decisions from page 206. The sixth design decision is not included as only Ben-Zvi's, Gadia's and Yeung's, and our algebras support transaction time. Gadia and Yeung associate transaction time with attribute values, Ben-Zvi associates transaction time with tuples, and we associate transaction time with relation states. Because several of the algebras have similar names and others are unnamed, we use the names of the developers to refer to the algebras hereafter for clarity. Table 8.1 categorizes the algebras according to their representation of valid time. Note that Clifford's algebra appears twice in Table 8.1 as it associates time-stamps with attributes in a relation scheme as well as tuples in a relation state (i.e., the tuple's lifespan). Table 8.2 describes other basic features of the types of objects defined and operations allowed in the algebras. The second column lists object types and the third column describes the structure of attributes. The fourth column indicates whether the algebras retain the set-theoretic semantics of the five basic relational operators or extend the operators to deal with time directly. The fifth column lists new operators introduced specifically to handle the temporal dimension of the phenomena being modeled.

In the next section we discuss a set of criteria for evaluating temporal extensions of the snapshot algebra. Then, in Section 8.5, we evaluate these 11 algebras against the criteria.

8.2 Criteria

Although several historical and temporal algebras have been proposed, previous research has not focused on defining criteria for evaluating the relative merit of these algebras. Only Clifford presents a list of specific properties desirable of a temporal extension of the snapshot algebra [Clifford & Tansel 1985]. He identifies five fundamental, conceptual goals, which will be discussed in detail shortly. These goals alone are insufficient to evaluate the relative

| Algebra | Objects | Attributes | Standard Operations | New Operations |
|---------------------|---|--|----------------------------------|--|
| Jones | historical states | atomic-valued | retained | time intersection, one-sided time intersection, time union, time difference, time-set membership |
| Ben-Zvi | snapshot states, temporal states | atomic-valued | extended | Time-View |
| Navathe & Ahmed | snapshot states, historical states | atomic-valued | retained | Time-Slice, Inner Time-View, Outer Time-View, TCJOIN, TCNJOIN |
| Sadeghi | historical states | atomic-valued | extended | Time Join, When |
| Sarda | snapshot states historical states | atomic-valued and non-atomic-valued | some retained others extended | Expand, Contract, Project-And-Widen, Concurrent Product |
| Lorentzos & Johnson | snapshot states | atomic-valued | retained | Extend, Fold, Unfold |
| Clifford & Croker | lifespans, historical states | functional | extended | When, Select-If, Select-When, Time-Slice, Time-Join |
| Gadia | temporal elements, historical states | functional | snapshot semantics | tdom, Temporal Selection |
| Gadia & Yeung | temporal states | functional | extended | Temporal Selection, Temporal Join |
| Tansel | historical states | atomic-valued set-atomic-valued triplet-valued set-triplet-valued | extended | Pack, Unpack, T-Dec, T-Form, Drop-Time, Slice, Uslice, Dslice, Enumeration |
| McKenzie | snapshot states, historical states | ordered pairs | extended | Temporal Derivation Temporal Aggregation |

Table 8.2: Objects and Operations in the Algebras

merit of the proposed algebras. A more comprehensive set of specific, objective criteria is needed. In this section, we identify 29 such criteria for evaluating temporal extensions of the snapshot algebra. First, we introduce the criteria. With each criterion, we indicate its source, if relevant. Next, we discuss our reasons for not including as criteria several other properties of historical and temporal algebras. Then, we examine incompatibilities among the criteria.

For clarity, we continue our convention of representing a historical operator as op to distinguish it from its snapshot algebra counterpart op .

Table 8.3 is an alphabetical listing of criteria for evaluating temporal extensions of the snapshot algebra. Included in this list are algebraic properties that have been advocated by others as well as those properties that seem reasonable to us. The list is restricted to only those properties that are well-defined, have an objective basis for being evaluated, and are arguably beneficial. No algebra can have all these properties as certain subsets of the properties are incompatible. An algebra can, however, have a maximal subset of properties from Table 8.3 that are compatible.

All attributes in a tuple are defined for the same interval(s) [Gadia 1986]. This requirement, termed *homogeneity* by Gadia, assumes that valid time is associated with attributes, rather than tuples, and that attributes are set-valued, rather than atomic-valued. Although attributes may change value at different times (i.e., asynchronous attributes), all attributes in a tuple must be defined for the same interval(s). Requiring that all attributes in a tuple be defined for the same interval(s) simplifies definition of the algebra. Operators need not be redefined to handle valid time directly. Rather, the algebra can be defined in terms of the conventional relational operators using snapshot semantics, even if set-valued attributes are allowed. Also, problems that arise when disjoint attribute time-stamps are allowed (e.g., how to handle non-empty time-stamps for some, but not all, attributes) need not be considered.

Consistent extension of the snapshot algebra [Clifford & Tansel 1985]. The expressive power of the algebra should subsume that of the snapshot algebra. The algebra should be at least as powerful as the snapshot algebra. Any relation or algebraic expression that can be represented in the snapshot model should have a counterpart in the temporal model. Thus the algebra should provide, as a minimum, a historical counterpart for each of the five operators that serve to define the snapshot algebra: union, difference, cartesian product, projection, and selection [Ullman 1982]. Furthermore, the historical relation state resulting from the application of one of these snapshot operators to a snapshot relation state and conversion of the resulting state to its historical counterpart should be equivalent to the historical relation state resulting from application of the snapshot operator's historical counterpart to the snapshot state's historical counterpart. If we assume that the function *Transform* transforms a snapshot state into its historical counterpart, then Figure 8.1 illustrates this equivalence proof.

Data periodicity is supported [Anderson 1982, Lorentzos & Johnson 1987A]. Periodicity is a property of many real-world phenomena. Rather than occurring just once in time or at randomly spaced times, these phenomena recur at regular intervals over a specific

- All attributes in a tuple are defined for the same interval(s)
- Consistent extension of the snapshot algebra
- Data periodicity is supported
- Each collection of valid attribute values is a valid tuple
- Each set of valid tuples is a valid relation state
- Formal semantics is specified
- Has the expressive power of a temporal calculus
- Historical data loss is *not* an operator side-effect
- Implementation exists
- Includes aggregates
- Incremental semantics defined
- Intersection, Θ -join, natural join, and quotient are defined
- Is, in fact, an algebra
- Model doesn't require *null* attribute values
- Multi-dimensional time-stamps are supported
- Optimization strategies are available
- Reduces to the snapshot algebra
- Restricts relation states to first-normal form
- Supports a three-dimensional visualization of historical states and operations
- Supports basic algebraic equivalences:

$$Q \dot{\cup} R \equiv R \dot{\cup} Q$$

$$Q \dot{\times} R \equiv R \dot{\times} Q$$

$$\dot{\sigma}_{F_1}(\dot{\sigma}_{F_2}(R)) \equiv \dot{\sigma}_{F_2}(\dot{\sigma}_{F_1}(R))$$

$$Q \dot{\cup} (R \dot{\cup} S) \equiv (Q \dot{\cup} R) \dot{\cup} S$$

$$Q \dot{\times} (R \dot{\times} S) \equiv (Q \dot{\times} R) \dot{\times} S$$

$$Q \dot{\times} (R \dot{\cup} S) \equiv (Q \dot{\times} R) \dot{\cup} (Q \dot{\times} S)$$

$$Q \dot{\times} (R \dot{-} S) \equiv (Q \dot{\times} R) \dot{-} (Q \dot{\times} S)$$

$$\dot{\sigma}_F(Q \dot{\cup} R) \equiv \dot{\sigma}_F(Q) \dot{\cup} \dot{\sigma}_F(R)$$

$$\dot{\sigma}_F(Q \dot{-} R) \equiv \dot{\sigma}_F(Q) - \dot{\sigma}_F(R)$$

$$\dot{\pi}_X(Q \dot{\cup} R) \equiv \dot{\pi}_X(Q) \dot{\cup} \dot{\pi}_X(R)$$

$$Q \dot{\cap} R \equiv Q \dot{-} (Q \dot{-} R)$$

- Supports relations of all four classes
- Supports scheme evolution
- Supports static attributes
- Supports rollback operations
- Treats valid time and transaction time orthogonally
- Tuples, not attributes, are time-stamped
- Unique representation for each historical relation state
- Unisorted (not multisorted)
- Update semantics is specified

Table 8.3: Criteria for Evaluating Temporal Extensions of the Snapshot Algebra

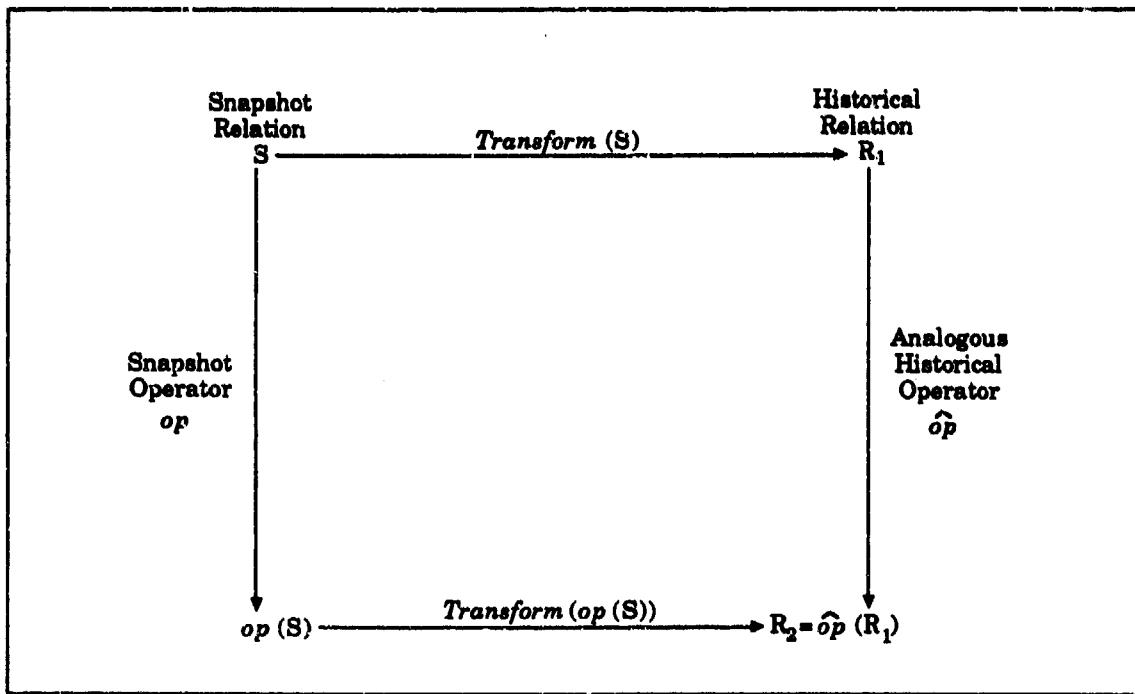


Figure 8.1: Outline of Equivalence Proof

interval in time. For example, a person may have worked from 8:00 a.m. until 5:00 p.m. each day, Monday through Friday, for a particular month. Ideally, a temporal data model should be able to represent such periodic phenomena without having to specify the time of each of their occurrences.

Each collection of valid attribute values is a valid tuple. In the snapshot model, the value of an attribute is independent of the value of other attributes in a tuple, except for key and functional dependency constraints. The same should be true of the temporal model. If we extend the snapshot model so that valid time is assigned to each attribute, we should extend the concept of attribute independence to include the valid-time component of the attribute as well as the value component of the attribute. Within a tuple, the value or valid-time component of one attribute shouldn't restrict arbitrarily the value or valid-time component of another attribute. Limiting valid tuples to some subset of the tuples that could be formed from valid attribute values adds a degree of complexity to the temporal model not found in the snapshot model.

Each set of valid tuples is a valid relation state. In the snapshot model, every set of tuples that satisfies value domain, key, and functional dependency constraints is a valid relation state. The same should be true of the temporal model. Imposing additional inter-tuple constraints, which further restrict the set of valid relation states, adds another degree of complexity to the temporal model not found in the snapshot model.

Formal semantics is specified. Concise, mathematical definitions for all object types and operations are needed. Without such definitions, the meaning of algebraic operations is unclear. Also, evaluation of the algebra is impossible.

Has the expressive power of a temporal calculus [Gadia 1986]. There should exist a temporal calculus whose expressive power is subsumed by that of the algebra. Calculus-based temporal query languages then can be developed for which the algebra can serve as the underlying evaluation mechanism.

Historical data loss is not an operator side-effect. Historical data are lost if an operator removes valid-time information, contained in underlying relation states, from its resulting relation state. Data loss becomes an operator side-effect if the removal of that valid-time information is not the purpose of the operator. For example, suppose a historical algebra allows attribute time-stamping but requires closure under Gadia's homogeneous restriction (i.e., the valid times associated with each attribute value in a tuple must be identical). To ensure closure under cartesian product, assume that cartesian product is defined using intersection semantics. Now consider the cartesian product of two historical relation states with attribute time-stamping, relation state A defined over the relation signature *Student* with attributes {*sname*, *course*}, and relation state B defined over the relation signature *Home* with attributes {*hname*, *state*}.

| A = | <i>sname</i> | <i>course</i> |
|-----|-----------------------|--------------------------|
| | ("Phil", {1, 3, 4}) | ("English", {1, 3, 4}) |

| B = | <i>hname</i> | <i>state</i> |
|-----|-----------------------|-------------------------|
| | ("Phil", {1, 2, 3}) | ("Kansas", {1, 2, 3}) |

| $A \hat{\times} B =$ | <i>sname</i> | <i>course</i> | <i>hname</i> | <i>state</i> |
|----------------------|--------------------|-----------------------|--------------------|----------------------|
| | ("Phil", {1, 3}) | ("English", {1, 3}) | ("Phil", {1, 3}) | ("Kansas", {1, 3}) |

Note the loss of valid-time information associated with Phil's enrollment in English at time 4 and his residency in Kansas at time 2. Algebras that allow such loss of historical data as an operator side-effect cannot support historical queries. If the algebra supports historical queries, the algebra must not allow loss of historical data as an operator side-effect; all valid-time information input to an operator must be preserved in the operator's output unless the operation being performed (e.g., difference, intersection) dictates removal.

Implementation exists. Semantic deficiencies, inconsistencies, and inefficiencies are often revealed during implementation. Therefore, it is desirable that the algebra have been implemented.

Includes aggregates. The temporal model should provide formal semantics for historical versions of standard aggregate (e.g., sum, count, min, max) operations.

Incremental semantics defined. Studies have shown that it may be more efficient to implement some recurring snapshot queries as incrementally maintained materialized views rather than recomputing the queries each time they are asked [Hanson 1987A, Hanson 1988, Roussopoulos 1987]. Because this strategy likely will be applicable to an even larger subclass of historical queries (c.f., Chapter 6), an incremental version of the algebra is needed if incremental maintenance of materialized views is to be supported.

Intersection, Θ -join, natural join, and quotient are defined. In the snapshot algebra, intersection, Θ -join, natural join, and quotient are defined in terms of the difference, selection, projection, and cartesian product operators [Ullman 1982]. In a historical algebra, analogous definitions may, but need not, hold. For example, if the historical versions of the basic operators don't retain the properties of their snapshot counterparts (e.g., satisfy algebraic equivalences), it may not be possible to define historical versions of intersection, Θ -join, natural join, and quotient exactly as they are defined in the snapshot algebra. Hence, formal definitions of these operators should be given.

Is, in fact, an algebra [Clifford & Tansel 1985]. This criterion is fundamental. Any algebra should define the types of objects supported and the allowable operations on object instances of each defined type. In addition, all legal operations should be closed.

Model doesn't require null attribute values. Restriction of attribute values to non-null values is consistent with the snapshot model and greatly simplifies the semantics of the algebra.

Multi-dimensional time-stamps are supported [Gadia & Yeung 1988]. It may be desirable to associate more than one aspect of time with an object or relationship being modeled. Because valid time, in particular, is a multifaceted aspect of time (c.f., Section 1.1.2), time-stamps of a single dimension may be inadequate for recording time in temporal databases. Hence, a temporal data model should support multi-dimensional time-stamps. Note that this criterion differs from the earlier one concerning periodicity. Satisfaction of the periodicity criterion only requires that the algebra support structured time-stamps that record a single aspect of time.

Optimization strategies are available. Except for semantics, implementation efficiency is the most important feature of an algebra. If an algebra cannot be implemented efficiently, it will have no practical application for the development of temporal query languages. Strategies for simplification of algebraic expressions corresponding to queries should be available. Note that the availability of basic algebraic equivalences already provides algebraic transformation optimizations.

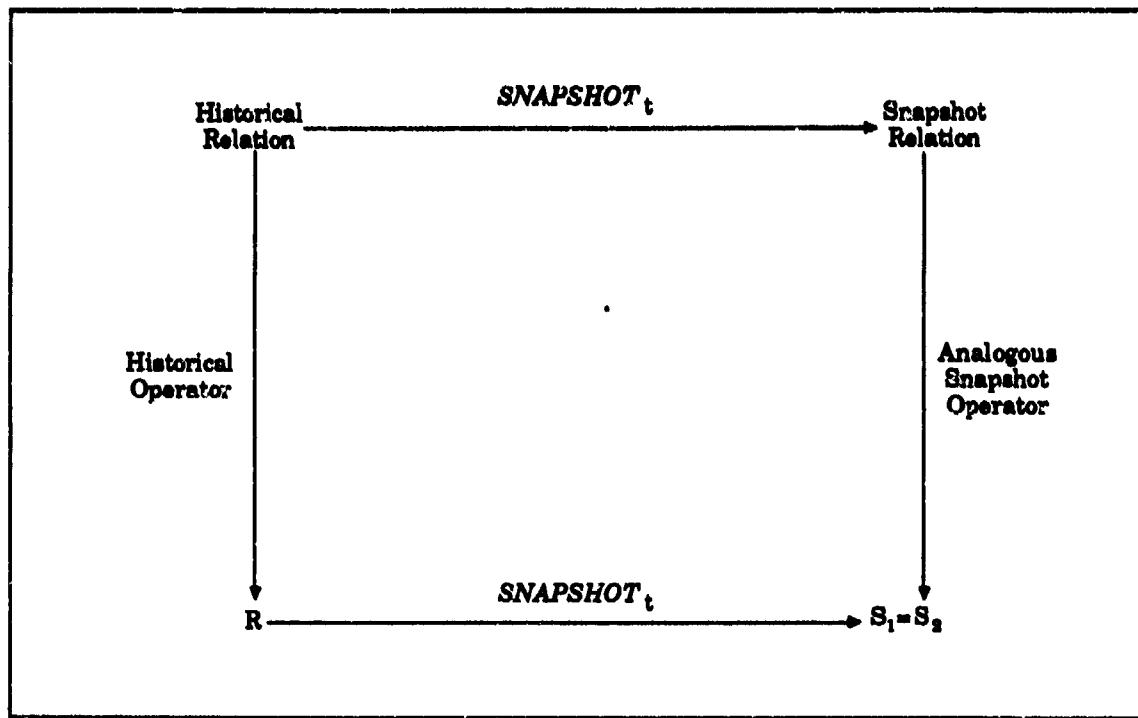


Figure 8.2: Outline of Reduction Proof

Reduces to the snapshot algebra [Snodgrass 1987]. The semantics of the algebra should be consistent with the intuitive view of a snapshot relation state as a two-dimensional slice of a three-dimensional historical relation state at a time t . Hence, for all historical operators, the snapshot state obtained by applying a historical operator to a historical state and then taking a snapshot should be equivalent to the relation state obtained by taking a snapshot of the historical state and applying the analogous relational operator to the resulting snapshot state. Figure 8.2 illustrates this reduction proof.

Restricts relation states to first-normal form. The snapshot algebra owes much of its simplicity to the restriction of relation states to first-normal form. Any extension of the snapshot algebra should retain this property.

Supports a three-dimensional conceptual visualization of historical states and operations [Ariav 1986, Ariav & Clifford 1986, Brooks 1956, Clifford & Tansel 1985]. Brooks was the first to propose that database relations recording changes to real-world objects over time be visualized conceptually as three-dimensional objects. Almost all proposals for extending the snapshot model to incorporate valid time are consistent with this “spatial metaphor” [Clifford & Tansel 1985], representing historical relation states as three-dimensional objects, whose third dimension is valid time. Although these spatial objects aren’t true cubes, they do possess geometric properties similar to those of cubes. For example, consider the historical state S_1 over the relation signature *Student* with attributes {*sname*, *course*} and attribute time-stamping.

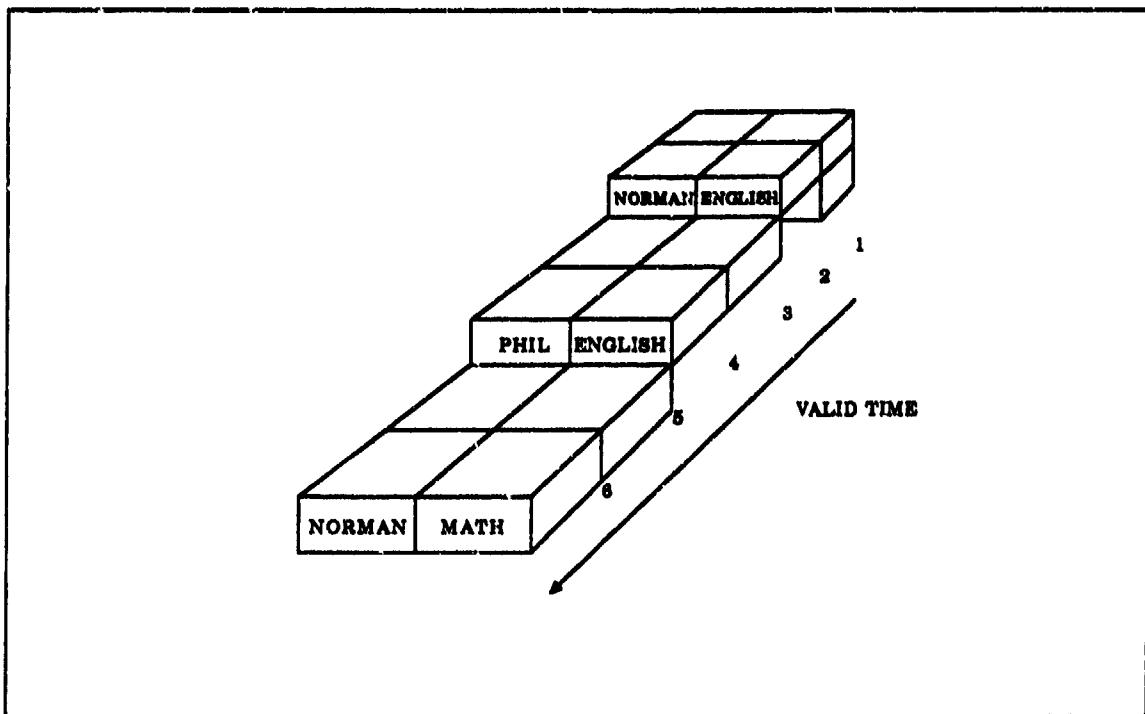


Figure 8.3: Historical Relation

| $S_1 =$ | student | course |
|-----------------------|--------------------------|--------|
| ("Phil", {1, 3, 4}) | ("English", {1, 3, 4}) | |
| ("Norman", {1, 2}) | ("English", {1, 2}) | |
| ("Norman", {5, 6}) | ("Math", {5, 6}) | |

Figure 8.3 is a graphical representation of this relation. Clearly, this representation of S_1 can be viewed as a three-dimensional object with geometric properties similar to that of a cube.

If we accept this three-dimensional representation as a high-level, user-oriented model of historical relation states, then each operation defined on historical relation states should have an interpretation, consistent with its semantics, in accordance with this conceptual framework. The definitions of operations should be consistent with the conceptual visualization that these operations manipulate spatial objects. For example, the difference operator should take two spatial objects (i.e., historical relation states) and produce a third spatial object that represents the volume (i.e., historical information) present in the first spatial object but not present in the second spatial object. Likewise, the cartesian product operator should take two spatial objects and produce a third spatial object such that each unit of volume (i.e., historical tuple) in the first spatial object is concatenated

with a unit of volume in the second spatial object to form a unit of volume in a third spatial object. This description of operations on historical relation states as "volume" operations on spatial objects is consistent not only with the conceptual visualization of historical relation states as three-dimensional objects but also with the semantics of the individual snapshot algebraic operations as "area" operations on two-dimensional tables, extended to account for the additional dimension represented by valid time.

Supports basic algebraic equivalences. The following commutative, associative, and distributive equivalences, which hold for and in some sense define the snapshot operators, should also hold for their historical counterparts.

$$Q \dot{\cup} R \equiv R \dot{\cup} Q$$

$$Q \dot{\times} R \equiv R \dot{\times} Q$$

$$\delta_{F_1}(\delta_{F_2}(R)) \equiv \delta_{F_2}(\delta_{F_1}(R))$$

$$Q \dot{\cup} (R \dot{\cup} S) \equiv (Q \dot{\cup} R) \dot{\cup} S$$

$$Q \dot{\times} (R \dot{\times} S) \equiv (Q \dot{\times} R) \dot{\times} S$$

$$Q \dot{\times} (R \dot{\cup} S) \equiv (Q \dot{\times} R) \dot{\cup} (Q \dot{\times} S)$$

$$Q \dot{\times} (R \dot{-} S) \equiv (Q \dot{\times} R) \dot{-} (Q \dot{\times} S)$$

$$\delta_F(Q \dot{\cup} R) \equiv \delta_F(Q) \dot{\cup} \delta_F(R)$$

$$\delta_F(Q \dot{-} R) \equiv \delta_F(Q) - \delta_F(R)$$

$$\pi_X(Q \dot{\cup} R) \equiv \pi_X(Q) \dot{\cup} \pi_X(R)$$

$$Q \dot{\cap} R \equiv Q \dot{-} (Q \dot{-} R)$$

Included in this list are the commutative, associative, and distributive equivalences involving only union, difference, and cartesian product in set theory [Enderton 1977]. Also included in this list are the non-conditional commutative laws involving selection and projection presented by Ullman [Ullman 1982]. Finally, the definition of the intersection operator in terms of the difference operator, which holds for the snapshot algebra, should also hold.

Supports relations of all four classes [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. As we saw in Chapter 1, relations may be classified, depending on their support for valid time and transaction time, as either snapshot, rollback, historical, or temporal relations. Any temporal extension of the snapshot algebra that supports both valid time and transaction time should allow for relations of all four classes.

Supports scheme evolution [Ben-Zvi 1982]. Because a relation's structure, as well as its contents, can change over time, a model of transaction time needs to support scheme evolution, as well as contents evolution.

Supports static attributes [Clifford & Tansel 1985, Navathe & Ahmed 1986]. The algebra should allow for attributes whose role in a tuple is not restricted by time. This

feature allows the temporal model to be applied to environments in which the values of certain attributes in a tuple are time-dependent while the values of other attributes in the tuple are not time-dependent.

Supports rollback operations [Ben-Zvi 1982, Snodgrass 1987]. In many database applications, there is a need to sometimes pose queries in the context of past database states. Hence, the algebra should allow relations to be rolled back to past states for query evaluation. The algebra should allow a query unrestricted access to tuples in past database states. Also, the algebra should allow a query access to multiple database states, rather than access to a single database state.

Treats valid time and transaction time orthogonally [Snodgrass & Ahn 1985, Snodgrass & Ahn 1986]. Valid time and transaction time are orthogonal aspects of time. Valid time concerns the time when events occur, and relationships exist, in the real world. Transaction time, on the other hand, concerns the time when a record of these events and relationships is stored in a database. Because the two aspects of time are orthogonal, their treatment also should be orthogonal. The valid time assigned to an object in the database shouldn't be restricted by or determined by the transaction time assigned that object. The algebra should allow both retroactive and postactive changes to be recorded. Also, operations involving one aspect of time shouldn't affect arbitrarily the other aspect of time.

Tuples, not attributes, are time-stamped. Time-stamping tuples, rather than attributes, simplifies the semantics of the algebra. Operators need not be defined to handle disjoint attribute time-stamps but rather can be defined in terms of the conventional relational operators using snapshot semantics.

Unique representation for each historical relation state. In the snapshot model, there is a unique representation for each valid snapshot relation state. Likewise, there should be a unique representation for each valid historical relation state. Failure of an algebra to satisfy this criterion can complicate the semantics of the operators, require inefficient implementations, and possibly restrict the class of database retrievals that can be supported. For example, consider the following relation states on the relation signature *Student* with attributes {*sname*, *course*} and attribute time-stamping.

| A = | <i>sname</i> | <i>course</i> |
|-----|--------------------|-----------------------|
| | ("Phil", {1, 2}) | ("English", {1, 2}) |
| | ("Phil", {3, 4}) | ("English", {3, 4}) |

| B = | <i>sname</i> | <i>course</i> |
|-----|--------------------------|-----------------------------|
| | ("Phil", {1, 2, 3, 4}) | ("English", {1, 2, 3, 4}) |

| C = | <table border="1"> <thead> <tr> <th>sname</th><th>course</th></tr> </thead> <tbody> <tr> <td>("Phil", {5, 6})</td><td>("English", {5, 6})</td></tr> </tbody> </table> | sname | course | ("Phil", {5, 6}) | ("English", {5, 6}) |
|--------------------|---|-------|--------|--------------------|-----------------------|
| sname | course | | | | |
| ("Phil", {5, 6}) | ("English", {5, 6}) | | | | |
| D = | <table border="1"> <thead> <tr> <th>sname</th><th>course</th></tr> </thead> <tbody> <tr> <td>("Phil", {2, 3})</td><td>("English", {2, 3})</td></tr> </tbody> </table> | sname | course | ("Phil", {2, 3}) | ("English", {2, 3}) |
| sname | course | | | | |
| ("Phil", {2, 3}) | ("English", {2, 3}) | | | | |

Clearly, the information content of states A and B is identical; the information content of state C is a continuation of the information in both A and B; and the information content of state D is a subset of that contained in both A and B. However, what is the semantics of $A \cup C$? Does the output relation state contain three tuples, two tuples, or just one tuple? Similarly, what is the semantics of $A \cup D$? Is the single tuple in D represented in the output relation state or is it absorbed by the two tuples in A? Also, if we want to retrieve the name of all students who were enrolled in English from time 2 to time 4, do we get the same result if we apply this query to relations A and B? Retrieval of "Phil," which is the intuitively correct result when applying this query to A, requires tuple selection based on information contained in more than one tuple, a significant departure from the semantics of the selection operation in the snapshot algebra. Thus, a selection operator with significantly more complicated semantics would be required to produce results that are correct intuitively. Moreover, the implementation of such a selection operator may be impractical because of the many cases that would have to be considered during the selection process.

Unisorted (not multisorted). In the snapshot algebra, all operators take as input and provide as output a single type of object, the snapshot relation state. If possible, a temporal extension of the snapshot algebra should also be unisorted. A multisorted algebra would introduce a degree of complexity in the temporal model not found in the snapshot model.

Update semantics is specified [Snodgrass 1987]. Concise, mathematical definitions for all update operations allowed on a relation's scheme as well as its contents are needed. Without such definitions, the meaning of update operations such as tuple insertion and tuple deletion is unclear.

8.3 Properties not Included as Criteria

The following properties are either subsumed by properties in Table 8.3, are not well-defined, or have no objective basis for being evaluated. Hence, they are not included as criteria.

Disallow tuples with duplicate attribute values. If attributes are time-stamped, then this requirement is subsumed by the criterion that the algebra have a unique representation for each historical relation state. There would be many different equivalent representations

for most historical states if tuples with duplicate attribute values were allowed. For example, the following are only two of several equivalent representations of a historical state A over the relation signature *Home* with attributes {*hname*, *state*} and attribute timestamping.

| $A =$ | <i>hname</i> | <i>state</i> |
|-------|----------------------------|--------------------------|
| | ("Norman", {1, 2, 5, 6}) | ("Utah", {1, 2, 5, 6}) |

| $A =$ | <i>hname</i> | <i>state</i> |
|-------|----------------------|--------------------|
| | ("Norman", {1, 2}) | ("Utah", {1, 2}) |
| | ("Norman", {5, 6}) | ("Utah", {5, 6}) |

Homogeneous tuples are valid tuples. This requirement is subsumed by the requirement that the algebra support non-homogeneous relations.

Supports historical queries (valid time) [Snodgrass 1987]. An algebra supports historical queries if information valid over a chronon can be derived from information in underlying relation states valid over other chronons, much as the snapshot algebra allows for the derivation of information about entities or relationships from information in underlying relation states about other entities or relationships. Satisfaction of this criterion implies that the algebra allows units of related information, possibly valid over disjoint chronons, to be combined into a single related unit of information possibly valid over some other chronon. Support for such a capability requires the presence, in the algebra, of a cartesian product or join operator that concatenates tuples, independent of their valid times, and preserves, in the resulting tuple, the valid-time information for each of the underlying tuples. Hence, this requirement is subsumed by the criteria that the algebra be a consistent extension of the snapshot algebra and historical data loss not be an operator side-effect.

Supports non-homogeneous relations [Gadia 1986]. If the algebra is closed and supports historical queries, it must support non-homogeneous relation states (i.e., relation states having tuples whose attribute values are allowed to have different valid times). Therefore, this requirement is subsumed by the criteria that the algebra, in fact, be an algebra, the algebra be a consistent extension of the snapshot algebra, and historical data loss not be an operator side-effect.

Treats valid time and transaction time uniformly [Gadia & Yeung 1988]. Gadia and Yeung have proposed a generalized relational model in which valid time and transaction time can be represented as two dimensions of a multi-dimensional attribute time-stamp [Gadia & Yeung 1988]. They also have shown how this uniform representation of valid time and transaction time can be used to advantage in expressing queries that involve changes in state. Ben-Zvi also has proposed a symmetrical representation for valid time and transaction time [Ben-Zvi 1982]. Unfortunately, this uniform treatment of valid time and transaction time can't be extended to include update operations. Transaction time has

a specific semantics, very different from that of valid time, that requires special handling on update. Valid time is specified by the user and its value can be derived, via an algebraic expression, from values in underlying relations. Transaction time, however, is simply the time, as measured by a system clock, when update occurs. Its value can't be specified by the user or derived from underlying relations. For update, therefore, it would seem impossible to treat valid time and transaction time uniformly and still retain a consistent semantics for transaction time. Hence, we don't include this property as a criterion.

Minimal extension of the snapshot algebra. This requirement is too vague to be considered a criterion, unless qualified. Criteria such as "consistent extension of the snapshot algebra," "reduces to snapshot algebra," and "unique representation for each historical relation state," all imply a minimal extension to the snapshot algebra.

Retains the simplicity of the snapshot model. Again, this requirement is too vague to be considered a criterion, unless qualified. Note that specific aspects of simplicity are implied by other properties that are well-defined (e.g., "model doesn't require *null* attribute values" and "algebra is unisorted").

The model is semantically complete [Clifford & Tansel 1985]. The model should serve as a standard for defining historical completeness (i.e., an extension of Codd's notion of completeness in the snapshot model). This requirement has no objective basis for evaluating models as there is no consensus definition of historical completeness.

8.4 Incompatibilities

Not all the criteria listed in Table 8.3 are compatible. There are certain subsets of criteria that no algebra can satisfy. In this section, we examine the incompatibilities among criteria.

The criterion that the algebra support a three-dimensional conceptual visualization of historical states and operations is incompatible with the criteria that

- Tuples, not attributes, be time-stamped,
- All attributes in a tuple be defined for the same interval(s), and
- The equivalence $Q \dot{\times} (R \dot{-} S) \equiv (Q \dot{\times} R) \dot{-} (Q \dot{\times} S)$ hold.

First, no algebra can support a three-dimensional conceptual model of historical states and operations and also time-stamp tuples. For the algebra to support a three-dimensional conceptual model of historical states and operations, the algebra must support a cartesian product or join operator that concatenates tuples, independent of their valid times, and preserves, in the resulting tuple, the valid-time information for each of the underlying tuples. Yet, if the cartesian product operator assigns different time-stamps to attributes in its output tuples, the criterion that tuples, not attributes, be time-stamped cannot be satisfied. Hence, no algebra can satisfy both of these criteria.

Secondly, no algebra can support a three-dimensional conceptual model of historical states and operations and also require that all attributes in a tuple be defined for the same interval(s). If the cartesian product operator required that all attributes in a resulting tuple be defined over the same interval(s), arbitrary valid-time information associated with the attributes of the underlying tuples could not be preserved and the criterion that the algebra support a three-dimensional conceptual model of historical states and operations could not be satisfied. Yet, if the cartesian product operator preserved the valid-time information for the attributes of the underlying tuples in the resulting tuple, attributes in the resulting tuple would be defined for different intervals and the criterion that all attributes in a tuple be defined for the same interval(s) could not be satisfied.

Thirdly, no algebra can support a three-dimensional conceptual model of historical states and operations and also support the distributive property of cartesian product over difference. For example, consider the following single-tuple historical states over the relation signature *Student* with attributes {*sname*, *course*} and attribute time-stamping.

| A = | <i>sname</i> | <i>course</i> |
|-----|-----------------------|-----------------------|
| | ("Phil", {1, 2, 3}) | ("Math", {1, 2, 3}) |

| B = | <i>sname</i> | <i>course</i> |
|-----|----------------------|-----------------------|
| | ("Norman", {1, 2}) | ("English", {1, 2}) |

| C = | <i>sname</i> | <i>course</i> |
|-----|-------------------|--------------------|
| | ("Norman", {2}) | ("English", {2}) |

Figure 8.4 illustrates the representation of historical states as spatial objects in calculating $A \hat{\times} (B - C)$ and $(A \hat{\times} B) - (A \hat{\times} C)$, respectively. The results of these calculations are shown below.

$$A \hat{\times} (B - C) =$$

| <i>sname</i> ₁ | <i>course</i> ₁ | <i>sname</i> ₂ | <i>course</i> ₂ |
|---------------------------|----------------------------|---------------------------|----------------------------|
| ("Phil", {1, 2, 3}) | ("Math", {1, 2, 3}) | ("Norman", {1}) | ("English", {1}) |

$$(A \hat{\times} B) - (A \hat{\times} C) =$$

| <i>sname</i> ₁ | <i>course</i> ₁ | <i>sname</i> ₂ | <i>course</i> ₂ |
|---------------------------|----------------------------|---------------------------|----------------------------|
| ("Phil", \emptyset) | ("Math", \emptyset) | ("Norman", {1}) | ("English", {1}) |

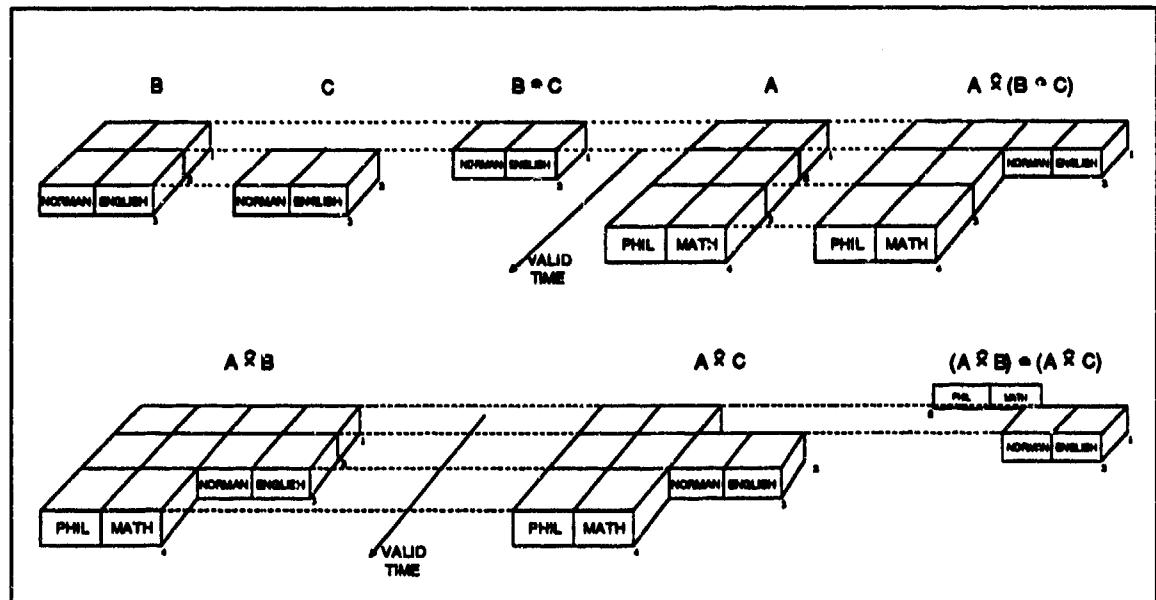


Figure 8.4: $A \hat{\times} (B - C)$ and $(A \hat{\times} B) - (A \hat{\times} C)$

This example shows that the criterion that the distributive property of cartesian product over difference hold is incompatible with the criterion that the algebra support a three-dimensional conceptual visualization of historical states and operations.

There are two other incompatibilities among the criteria in Table 8.3. First, the criterion that each set of valid tuples be a valid relation state is incompatible with the criterion that there be a unique representation for each relation state. If every set of valid tuples were allowed to be a valid relation state, the algebra could not have a unique representation for each state. For example, the following are only two of several equivalent representations of a relation state A over the relation signature *Home* with attributes {*hname*, *state*} and attribute time-stamping.

| $A =$ | <i>hname</i> | <i>state</i> |
|-------|----------------------------|--------------------------|
| | ("Norman", {1, 2, 3, 4}) | ("Utah", {1, 2, 3, 4}) |

| A = | hname | state |
|-----|----------------------|--------------------|
| | ("Norman", {1, 2}) | ("Utah", {1, 2}) |
| | ("Norman", {3, 4}) | ("Utah", {3, 4}) |

Yet, if the algebra allowed only one of these representations, there would be sets of valid tuples that would not be valid relation states. Hence, no algebra can satisfy both of these criteria.

Finally, the criteria that the algebra support a three-dimensional conceptual visualization of historical states and operations, have a unique representation for each historical relation state, and restrict relation states to first-normal-form are incompatible. An algebra can be defined that satisfies any two of these criteria, but no algebra can be defined that satisfies all three criteria. For example, consider the following two single-tuple relation states over the relation signature *Home* with attributes {hname, state} and attribute time-stamping.

| A = | hname | state |
|-----|-----------------------|-------------------------|
| | ("Phil", {1, 2, 3}) | ("Kansas", {1, 2, 3}) |

| B = | hname | state |
|-----|-----------------|-------------------|
| | ("Phil", {2}) | ("Kansas", {2}) |

To define difference so that $A - B$ can be calculated consistent with the conceptual model of historical operators as "volume" operators on spatial objects, the algebra must allow tuples with duplicate attribute values in a relation state

| $A - B =$ | hname | state |
|-----------|-----------------|-------------------|
| | ("Phil", {1}) | ("Kansas", {1}) |
| | ("Phil", {3}) | ("Kansas", {3}) |

or allow the time-stamp associated with a tuple to be non-atomic (i.e., a set of intervals rather than a single interval).

| $A - B =$ | hname | state |
|-----------|--------------------|----------------------|
| | ("Phil", {1, 3}) | ("Kansas", {1, 3}) |

Thus, to support a three-dimensional conceptual visualization of historical states and operations and disallow tuples with duplicate attribute values, which is implied by the criterion

| | | Supports a 3-dimensional view of historical states and operations? | |
|--|-----|--|--|
| | | No | Yes |
| Unique representation for each historical state? | No | No restrictions. | All attributes in a tuple cannot be defined over the same interval(s). The distributive property of cartesian product over difference cannot hold. Tuple time-stamping cannot be used. |
| | Yes | Each set of valid tuples cannot be a valid relation state. | All attributes in a tuple cannot be defined over the same interval(s). The distributive property of cartesian product over difference cannot hold. Tuple time-stamping cannot be used. Each set of valid tuples cannot be a valid relation. Relation states cannot be restricted to first-normal-form. |

Table 8.4: Incompatibilities Among Criteria

that the algebra have a unique representation for each historical state (if attributes are time-stamped), the algebra must allow non-first-normal-form relation states.

The five incompatibilities described above all involve at least one of the two criteria

- Supports a three-dimensional conceptual visualization of historical states and operations and
- Unique representation for each historical relation state.

Table 8.4 summarizes the effect satisfaction of these two criteria has on the algebra's ability to satisfy other criteria. Note that if the algebra satisfies neither of these criteria, then it can satisfy all the other criteria. If, however, the algebra satisfies both of these criteria, then there are five criteria that it cannot satisfy.

8.5 An Evaluation of Historical and Temporal Algebras

In this section we evaluate 11 algebras against the criteria presented in the previous section. We consider Ben-Zvi's Time Relational Model [Ben-Zvi 1982], Clifford's Historical Relational Data Model [Clifford & Croker 1987], Gadia's homogeneous model [Gadia 1988], Gadia's and Yeung's heterogeneous model [Gadia & Yeung 1988], Jones' extension to the

snapshot algebra to support time-oriented operations for LEGOL [Jones et al. 1979], Lorentzos' Temporal Relational Algebra [Lorentzos & Johnson 1987A], our algebra, Navathe's Temporal Relational Model [Navathe & Ahmed 1986], Sadeghi's historical algebra [Sadeghi 1987], Sarda's historical algebra [Sarda 1988], and Tansel's historical algebra [Tansel 1986]. Table 8.5 summarizes the evaluation of these 11 proposals against the criteria. We did not include TERM [Klopprogge 1981] and PDM [Manola & Dayal 1986], both of which include support for time, in this evaluation as they are temporal extensions of other data models. TERM is an extension of the Entity-Relationship model and PDM is an extension of the entity-oriented Daplex functional data model.

8.5.1 Conflicting Criteria

We first evaluate the algebras against the seven criteria introduced in the previous section that are not all compatible. Because no algebra can satisfy all seven of these criteria, we term the criteria *conflicting* criteria.

All attributes in a tuple are defined for the same interval(s). Only Gadia's homogeneous model satisfies this criterion. All the other algebras either time-stamp tuples or allow attribute time-stamps in a tuple to be disjoint.

Each set of valid tuples is a valid relation state. The algebras proposed by Ben-Zvi, Jones, Lorentzos, Sarda, and Tansel all satisfy this criterion. Gadia's homogeneous model also satisfies this criterion. Clifford's algebra fails to satisfy this criterion because a relation state can't have two tuples that match on the values of the key attributes at the same chronon. The heterogeneous model proposed by Gadia and Yeung, likewise, fails to satisfy this criterion; their algebra doesn't allow a relation state to have two tuples that match values on the key attributes. Our algebra also fails to satisfy this criterion because it doesn't allow relation states with *value-equivalent* tuples, that is, tuples with the same attribute values. Finally, the algebras proposed by Navathe and Sadeghi also fail to satisfy this criterion. Their algebras require that tuples with identical values for the explicit attributes be *coalesced*; hence, tuples with identical values for the explicit attributes can neither overlap nor be adjacent in time.

Restricts relation states to first-normal form. The algebras proposed by Ben-Zvi, Jones, Lorentzos, Navathe, and Sadeghi restrict relation states to first-normal form. The other algebras all fail to satisfy this criterion as they either allow set-valued attributes or set-valued time-stamps, or both.

Supports a three-dimensional conceptual visualization of historical states and operations. Our algebra supports the user-oriented conceptual visualization of a historical relation state as a three-dimensional object in that it supports non-homogeneous attribute time-stamping and prevents historical data loss as an operator side-effect. Operators in Clifford's algebra, with the exception of the join operators, satisfy this criterion. Although lifespans are associated with tuples, cartesian product is defined to prevent historical data loss as an operator side-effect through the introduction of nulls into the cartesian product's output tuples. It is unclear whether Gadia's and Yeung's algebra and Tansel's algebra

| Conflicting Criteria | Ben-Zvi | Clifford & Croker | Gadia | Gadia & Yeung | Jones et al. |
|--|---------|-------------------|-------|---------------|--------------|
| All attributes in a tuple are defined for same interval(s) | △ | △ | ✓ | △ | △ |
| Each set of valid tuples is a valid relation state | ✓ | △ | ✓ | △ | ✓ |
| Restricts relation states to first-normal form | ✓ | △ | △ | △ | ✓ |
| * Supports a 3-D view of historical states & operations | △ | P | △ | ? | △ |
| * Supports basic algebraic equivalences | ✓ | P | ✓ | ? | P |
| Tuples, not attributes, are time-stamped | ✓ | P | △ | △ | ✓ |
| * Unique representation for each historical relation state | △ | △ | △ | ✓ | △ |

| Compatible Criteria | Ben-Zvi | Clifford & Croker | Gadia | Gadia & Yeung | Jones et al. |
|--|---------|-------------------|-------|---------------|--------------|
| * Consistent extension of the snapshot algebra | ✓ | ✓ | ✓ | ? | ✓ |
| * Data periodicity is supported | △ | △ | △ | △ | △ |
| * Each collection of valid attribute values is a valid tuple | △ | △ | △ | △ | △ |
| * Formal semantics is specified | P | ✓ | ✓ | P | △ |
| * Has the expressive power of a temporal calculus | P | ? | ✓ | P | ? |
| * Historical data loss is not an operator side-effect | △ | P | △ | ? | △ |
| * Implementation exists | △ | △ | △ | △ | ✓ |
| * Includes aggregates | ✓ | △ | P | P | P |
| * Incremental semantics defined | △ | △ | △ | △ | △ |
| * Intersection, Θ-join, natural join, and quotient are defined | P | P | P | △ | △ |
| * Is, in fact, an algebra | ✓ | △ | ✓ | △ | ✓ |
| * Model doesn't require null attribute values | ✓ | △ | ✓ | ✓ | ✓ |
| * Multi-dimensional time-stamps are supported | △ | △ | △ | ✓ | △ |
| * Optimization strategies are available | ✓ | P | P | P | P |
| * Reduces to the snapshot algebra | ✓ | P | ✓ | P | ✓ |
| * Supports relations of all four classes | P | P | P | ✓ | P |
| * Supports schema evolution | △ | △ | △ | △ | △ |
| * Supports static attributes | △ | △ | △ | ✓ | △ |
| * Supports rollback operations | P | △ | △ | ✓ | △ |
| * Treats valid time and transaction time orthogonally | ✓ | ? | ? | ✓ | ? |
| * Unisorted (not multisorted) | △ | △ | △ | ✓ | ✓ |
| * Update semantics is specified | P | △ | △ | △ | △ |

✓ satisfies criterion

△ criterion not satisfied

P partial compliance

? not specified in papers

* maximal subset, see page 247

Table 8.5: Evaluation of Algebras Against Criteria

| Conflicting Criteria | Lorentzos & Johnson | McKenzie | Navathe & Ahmed | Sadeghi | Sarda | Tansei |
|--|---------------------|----------|-----------------|---------|-------|--------|
| All attributes in a tuple are defined for same interval(s) | △ | △ | △ | △ | △ | △ |
| Each set of valid tuples is a valid relation state | ✓ | △ | △ | △ | ✓ | ✓ |
| Restricts relation states to first-normal form | ✓ | △ | ✓ | ✓ | △ | △ |
| * Supports a 3-D view of historical states & operations | △ | ✓ | △ | △ | △ | ? |
| * Supports basic algebraic equivalences | ✓ | P | ? | ✓ | ? | P |
| Tuples, not attributes, are time-stamped | △ | △ | ✓ | ✓ | ✓ | △ |
| * Unique representation for each historical relation state | △ | ✓ | ✓ | ✓ | △ | △ |

| Compatible Criteria | ✓ | ✓ | ? | ✓ | ? | ? |
|--|---|---|---|---|---|---|
| * Consistent extension of the snapshot algebra | ✓ | ✓ | ? | ✓ | ? | ? |
| * Data periodicity is supported | ✓ | △ | △ | △ | △ | △ |
| * Each collection of valid attribute values is a valid tuple | △ | △ | △ | △ | ✓ | ✓ |
| * Formal semantics is specified | ✓ | ✓ | P | P | P | P |
| * Has the expressive power of a temporal calculus | ? | ✓ | P | P | P | ✓ |
| * Historical data loss is not an operator side-effect | ✓ | ✓ | ? | △ | ? | ? |
| * Implementation exists | ✓ | P | △ | ✓ | △ | △ |
| * Includes aggregates | ✓ | ✓ | △ | △ | △ | ✓ |
| * Incremental semantics defined | △ | ✓ | △ | △ | △ | △ |
| * Intersection, Θ-join, natural join, and quotient are defined | △ | ✓ | P | △ | △ | △ |
| * Is, in fact, an algebra | ✓ | ✓ | ✓ | ✓ | ? | ✓ |
| * Model doesn't require null attribute values | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| * Multi-dimensional time-stamps are supported | △ | △ | △ | △ | △ | △ |
| * Optimization strategies are available | P | P | P | P | P | P |
| * Reduces to the snapshot algebra | P | P | ✓ | ✓ | ✓ | P |
| * Supports relations of all four classes | P | ✓ | P | P | P | P |
| * Supports scheme evolution | △ | ✓ | △ | △ | △ | △ |
| * Supports static attributes | ✓ | ✓ | ✓ | ✓ | △ | ✓ |
| * Supports rollback operations | △ | ✓ | △ | △ | △ | △ |
| * Treats valid time and transaction time orthogonally | ? | ✓ | ? | ? | ? | ? |
| * Unisorted (not multisorted) | ✓ | △ | △ | ✓ | △ | ✓ |
| * Update semantics is specified | △ | ✓ | △ | △ | △ | △ |

✓ satisfies criterion

P partial compliance

△ criterion not satisfied

? not specified in papers

* maximal subset, see page 247

Table 8.5: Evaluation of Algebras Against Criteria (cont'd)

satisfy this criterion as all operations are not defined formally. The other algebras all fail to satisfy this criterion.

Lorentzos' algebra fails to satisfy this criterion when relation states have multiple attribute time-stamps. For example, consider the following single-tuple relations over the relation signature *Student* with attributes {*sname*, *course*}, valid in Lorentzos' algebra.

| A = | <i>sname</i> | <i>n-start</i> | <i>n-stop</i> | <i>course</i> | <i>c-start</i> | <i>c-stop</i> |
|-----|--------------|----------------|---------------|---------------|----------------|---------------|
| | "Marilyn" | 2 | 5 | "Math" | 2 | 5 |

| B = | <i>sname</i> | <i>n-start</i> | <i>n-stop</i> | <i>course</i> | <i>c-start</i> | <i>c-stop</i> |
|-----|--------------|----------------|---------------|---------------|----------------|---------------|
| | "Marilyn" | 1 | 4 | "Math" | 1 | 4 |

In Lorentzos' algebra, historical difference is defined in terms of the Unfold, set difference, and Fold operators. If we unfold both A and B, apply set difference to the unfolded relations, and then fold the result, we would get

| A - B = | <i>sname</i> | <i>n-start</i> | <i>n-stop</i> | <i>course</i> | <i>c-start</i> | <i>c-stop</i> |
|---------|--------------|----------------|---------------|---------------|----------------|---------------|
| | "Marilyn" | 2 | 4 | "Math" | 4 | 5 |
| | "Marilyn" | 4 | 5 | "Math" | 2 | 5 |

This result is inconsistent with the conceptual visualization of historical relation states as three-dimensional objects and operations on historical relation states as "volume" operations on spatial objects, as shown in Figure 8.5.

The homogeneous model proposed by Gadia and the algebras proposed by Ben-Zvi, Navathe, Jones, and Sadeghi also fail to satisfy this criterion. None of these algebras provides a cartesian product operator that allows for the concatenation of two tuples containing arbitrary historical information without the loss of historical information or, in Jones' algebra, the possible implicit addition of historical information. In Gadia's homogeneous model, attributes are time-stamped but the time-stamps of individual attributes are required to be identical. This requirement necessitates the definition of cartesian product using intersection semantics. In Ben-Zvi's algebra, tuples rather than attributes are time-stamped and a *Time Join* operator is defined using intersection semantics. Likewise, in Navathe's algebra, tuples rather than attributes are time-stamped and two operators, *TCJOIN* and *TCNJOIN*, are defined using intersection semantics. Navathe also defines two operators, *TJOIN* and *TNJOIN*, that allow for the concatenation of tuples without loss of historical information. These operators, however, are outside Navathe's algebra; they produce tuples with two time-stamps (R. Ahmed, personal communication, 1987). In Jones' algebra, tuples are time-stamped and cartesian product operators are defined using both intersection and union semantics. Finally, in Sadeghi's algebra, tuples are time-

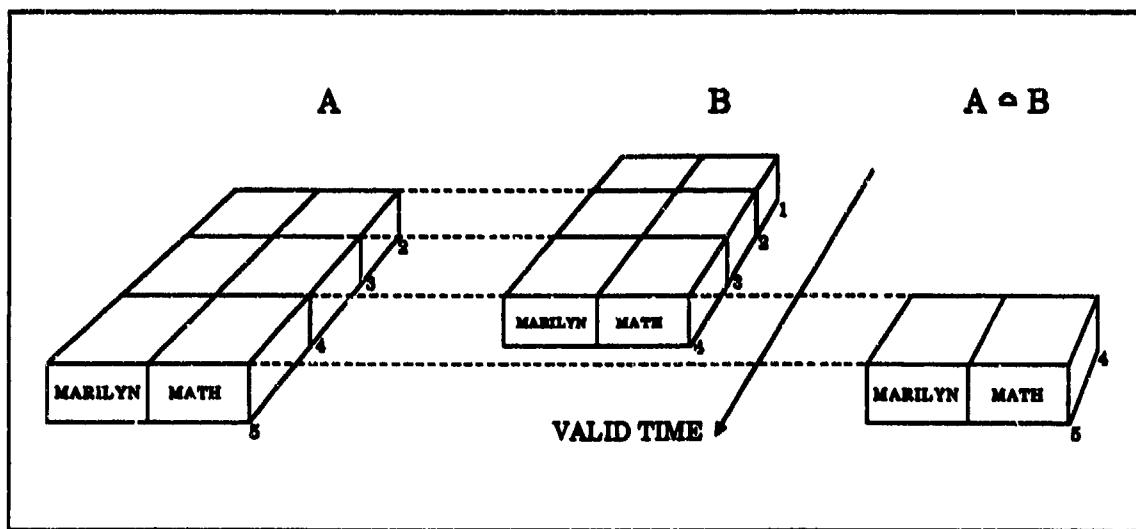


Figure 8.5: Conceptual View of the Difference Operator Applied to Historical Relations

stamped and the join and cartesian product operators are both defined using intersection semantics.

Consider the following single-tuple relation states over the relation signatures *Student* with attributes {*sname*, *course*} and *Home* with attributes {*hname*, *state*}. Assume attribute time-stamping.

| A = | <i>sname</i> | <i>course</i> |
|-----|--------------------------|-----------------------|
| | ("Marilyn", {2, 3, 4}) | ("Math", {2, 3, 4}) |

| B = | <i>hname</i> | <i>state</i> |
|-----|--------------------------|----------------------|
| | ("Marilyn", {1, 2, 3}) | (Texas, {1, 2, 3}) |

If cartesian product is represented conceptually as a "volume" operation on spatial objects, we would expect

$$A \hat{\times} B =$$

| <i>sname</i> | <i>course</i> | <i>hname</i> | <i>state</i> |
|--------------------------|-----------------------|--------------------------|----------------------|
| ("Marilyn", {2, 3, 4}) | ("Math", {2, 3, 4}) | ("Marilyn", {1, 2, 3}) | (Texas, {1, 2, 3}) |

as illustrated in Figure 8.6. However, since Gadia's homogeneous model and the algebras proposed by Ben-Zvi, Navathe, Jones, and Sadeghi all define cartesian product using intersection or union semantics, none can support this conceptual visualization of cartesian

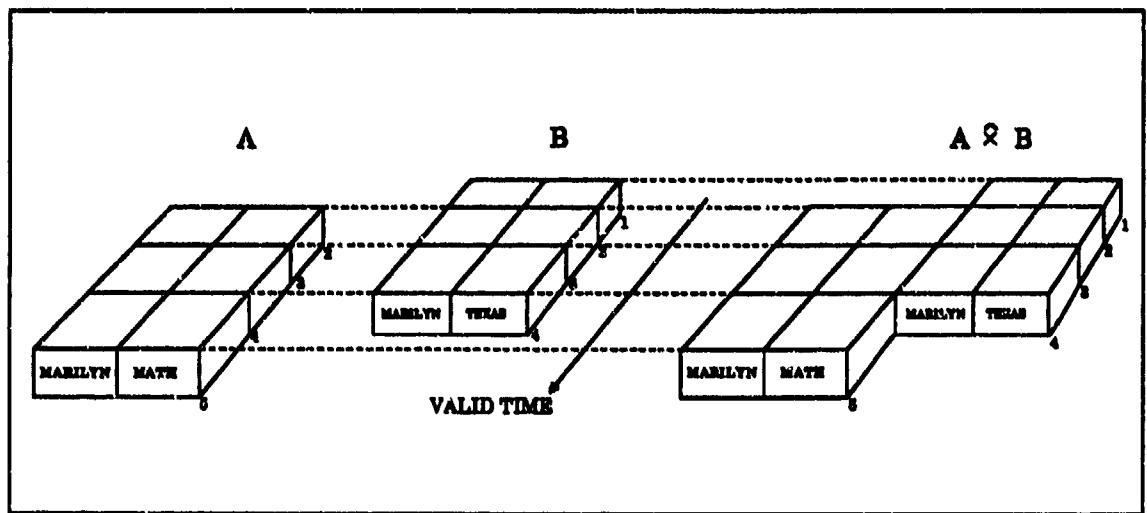


Figure 8.6: Cartesian Product of Historical Relations

product.

Sarda, in addition to defining a cartesian product operator using intersection semantics, allows the relational cartesian product operator to be applied to historical relation states. Although tuples in the result retain the time-stamps of their underlying tuples, the result is not a historical relation state. Its semantics is left unspecified. Hence, Sarda's algebra also fails to satisfy this criterion.

Supports basic algebraic equivalences. Ben-Zvi's algebra, Gadia's homogeneous model, and Lorentzos' and Sadeghi's algebras satisfy this criterion. Jones' algebra supports the equivalences, with one exception. The cartesian product operator defined using union semantics fails to support the distributive property of cartesian product over difference. All the equivalences, except the distributive property of cartesian product over difference, also hold for both Clifford's and our algebras. Tansel's algebra doesn't support the commutative property of selection with union and difference. It is unclear whether Tansel's algebra satisfies the other equivalences as union and difference are not defined formally. Similarly, it is unclear whether all the equivalences hold for Gadia's and Yeung's heterogeneous model, Navathe's algebra, and Sarda's algebra.

Tuples, not attributes, are time-stamped. Ben-Zvi, Jones, Navathe, Sadeghi, and Sarda all time-stamp tuples. Clifford also time-stamps tuples, but requires that the partial function from the time domain onto a value domain, representing an attribute's value, be further restricted to the attribute's time-stamp in the relation scheme. The other algebras all time-stamp attributes.

Unique representation for each historical relation state. Gadia's and Yeung's heterogeneous model supports a unique representation for each historical relation state because it doesn't allow two tuples to match values on the key attributes. Because our algebra al-

lows set-valued time-stamps and disallows value-equivalent tuples, it too supports a unique representation for each historical relation state. Because Navathe and Sadeghi require that value-equivalent tuples be coalesced, their algebras also satisfy this criterion. None of the other algebras satisfy this criterion. They all allow multiple representations of identical historical information within a relation state. Note that Clifford's algebra fails to satisfy this criterion because it only requires that no two tuples in a relation state match values on the key attributes *at the same chronon*. Hence, a relation state may contain value-equivalent tuples, even value-equivalent tuples adjacent in time, as long as they don't overlay in time.

8.5.2 Compatible Criteria

We now evaluate the algebras against the remaining 22 criteria. Because these criteria are compatible, an algebra can be defined that satisfies all these criteria.

Consistent extension of the snapshot algebra. Our algebra, along with those proposed by Ben-Zvi, Clifford, Jones, Lorentzos and Sadeghi, satisfy this criterion. Gadia's homogeneous model also satisfies this criterion. Although formal definitions for all operators are not provided for the other algebras, they too are likely to satisfy this criterion.

Data periodicity is supported. Only Lorentzos' algebra satisfies this criterion. Lorentzos' algebra allows multiple time-stamps of nested granularity, which can be used to specify periodicity. None of the other algebras allows multiple time-stamps of nested granularity.

Each collection of valid attribute values is a valid tuple. Tansel's and Sarda's algebras satisfy this criterion. Tansel's algebra time-stamps attributes without imposing any inter-attribute dependence constraints; any collection of valid attribute values is a valid tuple. Sarda's algebra encodes a tuple's time-stamp within a single attribute without imposing any inter-attribute dependence constraints.

The algebras proposed by Ben-Zvi, Jones, Navathe, and Sadeghi fail to satisfy this criterion because all three use implicit attributes to specify the end-points of a tuple's time-stamp, implicitly requiring that the value of the start-time attribute be less than (or " \leq ") the value of the stop-time attribute in all valid tuples. Lorentzos' algebra also requires that the values of attributes representing the boundary points of intervals be ordered. Clifford's algebra doesn't satisfy this criterion because the value of each attribute in a tuple is defined as a partial function from the time domain onto a value domain, where the function is restricted to times in the intersection of the tuple's time-stamp and the attribute's time-stamp in the relation scheme. Hence, the interval(s) for which an attribute is defined depends on both the tuple's time-stamp and the attribute's time-stamp in the relation scheme. Gadia's homogeneous model fails to satisfy this criterion because all attribute values in a tuple are required to be functions on the same temporal element. Gadia's and Yeung's heterogeneous model also fails to satisfy this criterion because relation states are restricted to non-null tuples. Finally, our algebra fails to satisfy this criterion because it doesn't allow the time-stamps of all attributes in a tuple to be empty.

Formal semantics is specified. We, Clifford, and Lorentzos provide a formal semantics for our algebras, as does Gadia for his homogeneous model. Jones, however, provides no

formal semantics for the time-oriented operations in LEGOL; she provides only a brief summary of time-oriented operations available in the language, along with examples illustrating the use of some of these operations.

Ben-Zvi and Tansel provide formal semantics for their algebras but provide incomplete definitions for certain operators. For example, Ben-Zvi's definition of the difference operator doesn't include a definition of the *Effective-Time-Start* and *Effective-Time-End* of tuples in the resulting relation, and Tansel doesn't provide formal definitions for his historical union and difference operators. Likewise, Gadia and Yeung don't provide formal definitions for their historical difference and intersection operators.

Navathe provides formal semantics for three new historical selection and four new historical join operators. He retains the five basic snapshot operators, although his model requires that value-equivalent tuples be coalesced. The semantics of these operators are left unspecified. Sadeghi also requires that value-equivalent tuples be coalesced. He provides formal semantics for all operators, but the semantics of some operators (e.g., union) doesn't preserve this value-equivalence property of historical relation states.

Sarda provides formal semantics for five new historical operators and selection and projection, when applied to historical relation states. Although he allows the cartesian product operator to be applied to historical states, he doesn't provide formal semantics for the result, which is not a historical state. He also doesn't provide formal definitions for historical union and difference.

Has the expressive power of a temporal calculus. Gadia has defined an equivalent calculus for his homogeneous model and we have shown, in Chapter 3, that our algebra has the expressive power of the TQuel calculus. Likewise, Tansel has defined an equivalent calculus for his algebra [Tansel & Arkun 1985]. Ben-Zvi has augmented the SQL Query Language with a *Time-View* operator and shown that the resulting language has expressive power equivalent to that of his algebra [Ben-Zvi 1982]. Rather than modify the semantics of the SQL Query Language to handle the temporal dimension, Ben-Zvi uses the *Time-View* operator as a temporal preprocessor to construct snapshot relations that can then be manipulated the same as any other snapshot relations. Yeung has defined an equivalent calculus for an earlier version of Gadia's and Yeung's heterogeneous model [Yeung 1986]. Navathe has defined the temporal query language TSQL [Navathe & Ahmed 1986], which is a superset of SQL, for use in his model. He has not shown, however, that his algebra has the expressive power of TSQL. Sadeghi has defined a historical query language HQL as an extension of the query language DEAL [Sadeghi 1987] and shown how to map queries in HQL onto expressions in his algebra. Sarda has extended SQL to handle historical queries and has shown how to map sample queries in this language onto expressions in his algebra [Sarda 1988]. A calculus has yet to be defined for any of the other proposed models.

Historical data loss is not an operator side-effect. Historical data loss is not an operator side-effect in our algebra. All operators are defined to retain, in their resulting relation states, the historical information found in their underlying relation states, unless removal is specifically required by the operator. Historical data loss also is not an operator side-effect in Lorentzos' algebra; all historical information is embedded in snapshot states and all

operations are defined in terms of the basic snapshot operators. In Clifford's algebra, all operators, with the exception of the join operators, are defined to prevent historical data loss as an operator side-effect. Ben-Zvi's algebra, Gadia's homogeneous model, and Jones' and Sadeghi's algebras all fail to satisfy this criterion because each time-stamps tuples and defines a cartesian product operator using intersection semantics. It is unclear whether the other algebras satisfy this criterion, as formal definitions for all operators are not provided.

Implementation exists. A prototype version of the algebra proposed by Jones has been implemented on the Peterlee Relational Test Vehicle [Jones et al. 1979]. Also, a prototype version of the algebra proposed by Lorentzos has been implemented on a PDP-11/44 as an extension of INGRES [Lorentzos & Johnson 1987B]. We have implemented a prototype of our algebra, without aggregates (c.f., Chapter 7). Sadeghi has developed an interpreter of his query language HQL [Sadeghi 1987]. To the best of our knowledge, implementations do not exist for the other algebras.

Includes aggregates. We along with Ben-Zvi define historical aggregate operators formally as part of our algebras. Tansel also defines historical aggregate functions in his algebra in terms of a new operator, termed *enumeration*, and an aggregate formulation operator [Tansel 1987]. Aggregate functions, defined for the snapshot algebra, can be used to compute historical aggregates in Lorentzos' algebra. The algebra proposed by Jones includes aggregate operators, but these operators are not defined formally. Although Gadia does not include aggregates in his models, he does introduce "temporal navigation" operators (e.g., First), which act similarly to the TQuel temporally oriented aggregates. The other algebras don't include any aggregate operators.

Incremental semantics defined. Our proposal satisfies this criterion; we define an incremental version of all operators in our algebra in Chapter 6. An incremental version of none of the other algebras is provided.

Intersection, Θ -join, natural join, and quotient are defined. Historical versions of these four operators are defined for our algebra (c.f., Section 3.6). Ben-Zvi defines a join operator, and Clifford defines intersection, Θ -join, and natural join operators. Gadia defines intersection, Θ -join, and natural join in his homogeneous model. Yeung defines all four operators in an earlier version of Gadia's and Yeung's heterogenous model [Yeung 1986], but they aren't defined in the later version of this model [Gadia & Yeung 1988]. Finally, Navathe defines historical versions of join and natural join. None of the other algebras defines historical versions of these operators.

Is, in fact, an algebra. Clifford's algebra fails to satisfy this criterion because it is not closed under union, difference, or intersection. The historical versions of these binary operators are defined for two relation states only if they are *merge compatible* (i.e., tuples from the two relation states that match on the values of the key attributes at some chronon must also match on all their attribute values at each chronon in the intersection of their lifespans). Likewise, Gadia's and Yeung's heterogeneous model doesn't satisfy this criterion because it is not closed under union. The union of two relation states is undefined if there are tuples in the relation states that match on the values of the key attributes but have different values at some time for some attribute. It is unclear whether Sarda's proposal

satisfies the closure property as cartesian product of historical states, although allowed, produces a result that is not a historical state. Its semantics, however, is left unspecified. Each of the other proposals being evaluated satisfies this criterion.

Model doesn't require null attribute values. Clifford's algebra fails to satisfy this criterion. The cartesian product operator assigns null values to attributes in an output tuple for each chronon that is in the lifespan of the output tuple but not in the lifespan of the input tuple associated with that attribute. The other algebras being evaluated all satisfy this criterion.

Multi-dimensional time-stamps are supported. Only Gadia's and Yeung's heterogeneous model satisfies this criterion. None of the other algebras supports multi-dimensional time-stamps. We discuss, however, extension of our algebra to support multi-dimensional time-stamps in Section 3.6.

Optimization strategies are available. Ben-Zvi describes an efficient implementation of his algebra, while Gadia presents a computational semantics, designed to aid efficient implementation of the algebra, for his homogeneous model. Also, optimization techniques based on the algebraic equivalences, with certain exceptions for some algebras, could be used in an implementation of any of the 11 algebras.

Reduces to the snapshot algebra. Gadia's homogeneous model satisfies this criterion; operators are defined using a snapshot semantics thus guaranteeing that the algebra reduces to the snapshot algebra. Likewise, the descriptions of the algebras proposed by Ben-Zvi and Jones imply that the operators are defined using snapshot semantics. Because Navathe, Sadeghi, and Sarda all assume tuple time-stamping, their algebras also satisfy this criterion. Although formal definitions have not been provided for all operators in Gadia's and Yeung's heterogeneous model, the algebra can satisfy this criterion only through the introduction of distinguished *null's* when taking snapshots. Because we, along with Tansel and Lorentzos, allow non-homogeneous attribute time-stamps, our algebras also satisfy this criterion only through the introduction of distinguished *null's* when taking snapshots. Likewise, because Clifford doesn't require that all attributes in a tuple be defined for the same lifespan (i.e., an attribute's value in a tuple is specified only for chronons in the intersection of the tuple's lifespan and the attribute's lifespan in the relational scheme), his algebra also satisfies this criterion only through the introduction of distinguished *null's* when taking snapshots.

Supports relations of all four classes. Gadia's and Yeung's heterogeneous model, because it allows multi-dimensional time-stamps, can support relations of all four classes. Our algebra also satisfies this criterion. Ben-Zvi's model, although it supports both valid time and transaction time, can support rollback and historical relation only by embedding them in temporal relations. The other algebras, since they don't support transaction time, can't support rollback or temporal relations.

Supports scheme evolution. Our algebra satisfies this criterion. Ben-Zvi, while describing an approach for representing an evolving scheme as a temporal relation, doesn't include provisions for scheme evolution in the formal semantics of his algebra. Hence, his algebra fails to satisfy this criterion. Gadia and Yeung, although they support transaction

time, don't address the problem of scheme evolution. Martin describes an approach for handling scheme changes in Navathe's formalization [Martin et al. 1987], but the algebra is not extended to support scheme evolution. Because the other algebras don't support transaction time, they too fail to satisfy this criterion.

Supports static attributes. Lorentzos', Navathe's, Sadeghi's and Tansel's algebras satisfy this criterion by allowing both time-dependent and non-time-dependent attributes. Our algebra and Gadia's and Yeung's heterogeneous model also can support static attributes. In these two algebras, the time-stamp of an attribute can be defined independently of the time-stamps of any of the other attributes in a tuple. In our algebra we would represent a static attribute as an attribute assigned the time domain. Clifford's algebra fails to satisfy this criterion because an attribute's value in a tuple can't be specified for chronons that aren't in the tuple's lifespan. The other four algebras all require that the same valid time be associated with all attributes in a tuple; therefore, none of these algebras can support static and time-dependent attributes within the same tuple.

Supports rollback operations. Our algebra satisfies this criterion. The rollback operators allow queries to be posed on one, or more, arbitrary relation states without restriction on the tuples in those states that participate in the query. Gadia's and Yeung's algebra also satisfies this criterion; transaction time is treated simply as another dimension in a multi-dimensional temporal element. Ben-Zvi's algebra, although it allows rollback, achieves only partial satisfaction of this criterion because it requires that the tuples participating in a query all have a specified valid time in common. All operations in Ben-Zvi's algebra are defined in terms of a transaction time t_s and a valid time t_e . During expression evaluation, rollback occurs to the relation state at t_s , but only tuples valid at t_s are accessed. None of the other algebras supports rollback operations.

Treats valid time and transaction time orthogonally. Our algebra, along with that proposed by Ben-Zvi, satisfies this criterion. Gadia's and Yeung's heterogeneous model also satisfies this criterion. All three support retroactive and postactive changes and allow independent assignments of valid time and transaction time, without restrictions. The other algebras all fail to satisfy this criterion because they don't support transaction time.

Unisorted (not multisorted). The algebras proposed by Jones, Lorentzos, Sadeghi, and Tansel and the heterogeneous model proposed by Gadia and Yeung are unisorted in that they define only one object type. All the other algebras are multisorted. We define algebraic operators on snapshot states and historical states. Gadia's homogeneous model is a multisorted algebra; its object types are historical relation states and temporal expressions. Clifford defines a multisorted algebra whose object types are historical relation states and lifespans. Ben-Zvi allows both snapshot and temporal relation states while Navathe allows both snapshot and historical relation states. Finally, Sarda defines a projection operator that is allowed to map a historical state onto a snapshot state.

Update semantics is specified. Our proposal satisfies this criterion; the semantics of update are formalized in Chapter 4. Ben-Zvi defines the semantics of tuple insertion, deletion, and modification but does not extend the formalization to include scheme evolution. The other proposals do not consider update semantics in their formalizations.

Supports a 3-dimensional view of historical states and operations?

| | | No | Yes |
|--|-----|---|-----------------------------|
| | | No | Clifford & Croker Tansel |
| Unique representation for each historical state? | No | Ben-Zvi Gadia Jones, et al. Lorentzos & Johnson Sarda | |
| | Yes | Navathe & Ahmed Sadeghi | Gadia & Yeung McKenzie |

Table 8.6: Classification of Algebras According to Criteria Satisfied

8.5.3 Evaluation Summary

Of the 29 criteria listed in Table 8.3, each is satisfied by at least one of the 11 algebras and three are satisfied, at least partially, by all the algebras. As was shown in Table 8.4, the subset of conflicting criteria that an algebra can satisfy is necessarily dependent on whether the algebra supports a three-dimensional conceptual visualization of historical states and operations and whether each historical relation in the algebra has a unique representation. For example, we, Gadia and Yeung, Navathe, and Sadeghi cannot satisfy the criterion that each set of valid tuples is a valid relation because our algebras satisfy the criterion that each historical relation has a unique representation. In Table 8.6 all 11 algebras are classified according to their satisfaction of these two criteria. (We assume, for purposes of discussion, that operators not defined by Gadia and Yeung and by Tansel could be defined consistent with the conceptual visualization of historical relation states as spatial objects and operations on historical relation states as “volume” operators on spatial objects). According to this classification and the summary of incompatibilities among criteria in Table 8.4, Navathe’s and Sadeghi’s algebras can’t satisfy one of the remaining conflicting criteria, Clifford’s and Tansel’s algebras can’t satisfy three of the remaining criteria, while our algebra and the heterogeneous model proposed by Gadia and Yeung can’t satisfy any of the remaining conflicting criteria. The other algebras are not restricted from satisfying the remaining conflicting criteria. There is no apriori reason any of the compatible criteria cannot be satisfied; one measure of the quality of the design of an algebra is the extent to which it satisfies these criteria.

As no algebra can satisfy all the criteria, a ranking is necessary to identify a maximal

subset of the criteria. Of the conflicting criteria, we consider the criterion that an algebra support a three-dimensional conceptual visualization of historical states and operations to be the most important. If an algebra fails to support this criterion, its semantics is inconsistent with user intuition for operations on historical relation states. Hence, we don't include the criteria that

- Tuples, not attributes, be time-stamped,
- All attributes in a tuple be defined for the same interval(s), or
- The equivalence $Q \dot{\times} (R - S) \equiv (Q \dot{\times} R) - (Q \dot{\times} S)$ hold

in the maximal subset of criteria as they all conflict with the criterion that the algebra support a three-dimensional conceptual visualization of historical states and operations. We do, however, include all the other algebraic equivalences. Of the three conflicting criteria that remain, we consider the criterion that there be a unique representation for each relation state more important than the criterion that each set of valid tuples be a valid relation state or the criterion that relation states be restricted to first-normal-form. Only by requiring that each relation state have a single representation can we define and implement algebraic operators with consistent semantics in terms of tuple membership in a set-theoretic relation state rather than in terms of multiple-element relation-state equivalence classes. Hence, we propose as maximal the subset of criteria containing the compatible criteria from Table 8.5 and the criteria that

- The algebra support a three-dimensional conceptual visualization of historical states and operations,
- There be a unique representation for each historical state, and
- All the equivalences from Table 8.3, except for the distributive property of cartesian product over difference, hold.

These are indicated by an "*" in Table 8.5 on pages 236 and 237.

Our algebra satisfies this maximal subset of criteria, either fully or partially, with four exceptions. First, our algebra doesn't support periodicity. However, as we pointed out in Section 3.6, our algebra can be extended to support periodicity by allowing structured time-stamps. Second, our algebra doesn't support multi-dimensional time-stamps. Again, we discuss extension of the algebra to support multi-dimensional time-stamps in Section 3.6. Third, our algebra doesn't allow each collection of valid attribute values to be a valid tuple; we require that the valid-time component of at least one attribute in each tuple be non-empty. Fourth, our algebra is multisorted. None of the other algebras reviewed here achieves these results.

8.6 Review of Design Decisions

Having identified a maximal subset of evaluation criteria for temporal extensions of the snapshot algebra, we can now explain our choices to the design decisions listed on page 206. To motivate our choices to the design decisions, we emphasize the importance of those choices in determining the properties of the algebra.

8.6.1 Time-stamped Attributes

We chose to time-stamp attributes rather than tuples to support historical queries. Support for historical queries required that we define a cartesian product operator that concatenates tuples, independent of their valid times, and preserves, in the resulting tuple, the valid-time information for each of the underlying tuples. Only by time-stamping attributes could we define a cartesian product operator with this property and maintain closure under cartesian product.

8.6.2 Set-valued Time-stamps

We chose to allow set-valued attribute time-stamps to support a three-dimensional conceptual visualization of historical states and operations, satisfy various algebraic equivalences, ensure a unique representation for each relation state, and prevent temporal information loss as an operator side-effect. If we had decided to disallow set-valued attribute time-stamps, then we would have had to have permitted value-equivalent tuples to model accurately real-world temporal relationships. Yet, value-equivalent tuples, because they spread temporal relationships among attributes across tuples, would have caused problems in defining an algebra with the above properties. If value-equivalent tuples had been allowed (and set-valued attribute time-stamps disallowed), a unique representation for each historical relation could not have been specified without imposing inter-tuple restrictions on the attribute time-stamps of value-equivalent tuples. Also, historical operators, in particular the difference operator, that would have satisfied both the algebraic equivalences and the conceptual visualization of historical operations as "volume" operations on spatial objects, while preventing loss of information about temporal relationships as an operator side-effect, could not have been defined.

8.6.3 Single-valued Attributes

We chose to restrict attributes to single values to retain in our algebra the commutative properties of the selection operator found in the snapshot algebra. If we had allowed set-valued attributes, without imposing intra-tuple restrictions on attribute time-stamps, then we would have had to have combined the functions of the selection and historical derivation operators into a single, more powerful operator. This consolidation would have been necessary to ensure that the temporal predicate in the current historical derivation operator was considered to be true for an assignment of intervals to attribute names only when the

predicate in the current selection operator held for the attribute values associated with those intervals. This new operator would have satisfied the commutative properties of the current selection operator only in restricted cases. Hence we would have limited the usefulness of key optimization strategies in future implementations of our algebra.

8.6.4 Extended Operator Semantics

We chose to extend the semantics of the conventional relational operators to handle the temporal dimension directly to support a three-dimensional conceptual visualization of historical states and operations and ensure a unique representation for each relation state. Retention of the set-theoretic semantics of the operators would have prevented the algebra from satisfying these criteria. We defined the semantics of the historical version of each snapshot operator to be a consistent extension of the snapshot operator's semantics. Hence, each expression in the snapshot algebra has an equivalent counterpart in the historical algebra and expressions in the historical algebra reduce to their snapshot counterparts when all attribute time-stamps are the same. Also, we defined all operators to prevent loss of temporal information as an operator side-effect.

8.6.5 New Temporal Operators

We chose to handle temporal selection, projection, and aggregation by introducing new operators to perform these functions. We would have preferred separate operators for temporal selection and projection, but were forced to include both functions in the derivation operator because we chose to allow set-valued attribute time-stamps. If we had disallowed set-valued time-stamps, we could have replaced the derivation operator by two simpler operators, analogous to the selection and projection operators, that would have performed tuple selection and attribute projection in terms of the valid-time components, rather than the value components, of attributes. But, as we discussed above, disallowing set-valued time-stamps would have required that the algebra support value-equivalent tuples, which would have prevented the algebra from having several other, more highly desirable properties.

8.6.6 Transaction Time and Relation States

We chose to assign transaction time to relation states rather than tuples or attributes. In so doing, we were able to separate, almost entirely, consideration of valid time and transaction time in defining the semantics of our algebra. Except for the rollback operators, all operators, both snapshot and historical, were defined independently of any consideration of transaction time. Similarly, we were able to define the semantics of update, rollback, and scheme evolution, without change to the snapshot operators and their historical counterparts. Our algebra is consistent with the conceptual visualization of snapshot and historical relations as single-state relations and rollback and temporal relations as multiple-state relations, indexed by transaction time. The algebra also is consistent with the conceptual

visualization of database update as change in relation states. Finally, by assigning transaction time to relation states and valid time to attributes, we emphasized the orthogonality of the two aspects of time.

8.7 Summary

In this chapter we have motivated the choices we made to the design decision listed on page 206. In so doing, we evaluated 11 temporal extensions of the snapshot algebra against 29 criteria. We first described the algebras in terms of the types of objects they define and the operations on object instances they allow. Then, we introduced evaluation criteria, each of which is well-defined, has an objective basis for being evaluated, and is arguably beneficial. We omitted properties from the list of criteria that were either subsumed by criteria, not well-defined, or had no objective basis for being evaluated. We also identified incompatibilities among the criteria. Finally, we evaluated the algebras against the criteria, proposed a maximal subset of the criteria, and reviewed our design decisions, considering our goal to define an algebra that satisfies as many desirable properties as possible. Our algebra satisfies all but three of the criteria in the maximal subset of criteria. None of the other algebras reviewed here achieves these results.

Chapter 9

Conclusions and Future Work

The thesis of this research is that the snapshot algebra can be extended to support query and update of temporal databases, while also accommodating the incremental update of materialized views. To prove this thesis we defined an algebraic language for query and update of temporal databases. In this chapter we summarize the contributions of this work, draw some conclusions, and discuss possible areas of future research. This summary augments that found at the end of each preceding chapter.

9.1 Contributions

We have investigated extension of the snapshot algebra to support two aspects of time: valid time and transaction time. We have identified design decisions and problems that arise when one attempts to extend the snapshot algebra to support time and have posed solutions to those design decisions and problems. Because the snapshot algebra is a significant component of the relational data model, our work helps to determine the applicability and extendibility of the relational data model to a temporal data model. Our work also is an important step toward implementation of a temporal data model, as it is compatible with many of the existing optimization techniques used to implement RDBMS's. A brief description of specific contributions of this research follows.

9.1.1 Language

The language, itself, is the major contribution of this work. It is a consistent and comprehensive extension of the snapshot algebra for dealing with valid time and transaction time. The expressive power of the snapshot algebra for database query and update is subsumed by that of the language. Also, the language

- Formalizes both query and update of temporal databases;
- Treats valid time and transaction time orthogonally;

- Supports databases containing snapshot, rollback, historical, and temporal relations;
- Accommodates both scheme and contents evolution;
- Handles multiple-command, as well as single-command, transactions;
- Supports queries on valid time;
- Allows relations to be rolled back to a previous transaction time;
- Supports both unmaterialized and materialized views; and,
- Accommodates a spectrum of view maintenance strategies, including query modification, in-line view evaluation, immediate recomputation, and immediate incremental update.

Both the syntax and semantics of the language are defined formally. A variant of Backus-Naur Form is used to specify the syntax; denotational semantics is used to specify the semantics. Formal definitions are given for the types of objects and the operations on object instances allowed in the language. These formal definitions serve as the basis for proving that the language has the expressive power of calculus-based query languages. Also, because the language's semantics is defined in a rigorous and formal way, implementations may be checked against it and proven correct.

The language is shown to have the expressive power of the temporal query language TQuel. The algebraic equivalence of each TQuel statement is given. The TQuel retrieve statement, without aggregates and with aggregates in its target list, where clause, and when clause, is considered, as are the create, append, delete, and replace modification statements. Hence, the language has sufficient expressive power to serve as the underlying evaluation mechanism for TQuel.

9.1.2 Temporal Algebra

Definition of a temporal algebra is another contribution. Formal definitions for 13 operators are given, and the definition of each operator is consistent with the user-oriented conceptual visualization of historical relation states as three-dimensional objects. Nine of the operators have counterparts in the snapshot algebra (i.e., union, difference, cartesian product, selection, projection, intersection, Θ -join, natural join, and quotient) and one, historical derivation, effectively performs selection and projection on the valid-time, rather than the value, component of attributes. Two others support unique and non-unique historical aggregation. Both of these aggregate operators are defined to accommodate aggregation windows of arbitrary width as well as families of arbitrary scalar aggregate functions. The last operator, rollback, allows relations to be rolled back to a previous transaction time. Although the algebra is a relatively straightforward extension of the snapshot algebra, it has a collection of desirable properties satisfied in concert by no other temporal algebra. Also, it is consistent with the snapshot algebra; the semantics of each operator having a snapshot counterpart reduces to that of its snapshot counterpart when valid time is held constant.

9.1.3 Incremental Temporal Algebra

A third contribution of this research is definition of an incremental version of our temporal algebra. The incremental algebra serves, via incremental expression evaluation, as the basis for incremental update of materialized historical views. An incremental version of each of the 13 operators in the temporal algebra is defined. In defining the incremental temporal algebra we show that our temporal algebra is as amenable to the incremental update of materialized views as is the snapshot algebra.

9.1.4 Prototype Implementation

Another contribution of this research is a prototype implementation of an incremental query processor for TQuel. The prototype treats query plans as view definitions for materialized views, where the views are maintained via the incremental temporal algebra. In building the prototype, we show that a standard architecture for incremental update of materialized views in snapshot databases can be adapted to incremental update of materialized views in temporal databases. We also show that the incremental temporal algebra is compatible with known optimization techniques for implementing incremental query processors; optimization techniques used to implement incremental query processors for non-temporal query languages apply equally to our incremental query processor for TQuel.

9.1.5 Evaluation Criteria

The final contribution of this research is identification of criteria for evaluating temporal extensions of the snapshot algebra. A set of 29 such criteria are presented. These criteria, although not all compatible, are well-defined, have an objective basis for being evaluated, and are arguably beneficial. Twenty-five of the criteria are proposed as the maximal subset of mutually compatible criteria that a temporal algebra could support. In addition to serving as the basis for objective evaluation of different temporal algebras, the criteria can be used as a guide in making design decisions when defining a temporal algebra that will result in an algebra with a maximal subset of desirable properties. To our knowledge, there has been no previous attempt to identify a comprehensive set of well-defined, objective criteria for judging the relative merit of temporal extensions of the snapshot algebra.

Ten different proposals for extending the snapshot algebra to support some aspect of time are described in terms of the types of objects they define and the operations on object instances they allow. These proposals, along with our language, are evaluated against the 25 criteria we propose as a maximal subset of criteria. Our language satisfies all but three of the criteria. None of the other proposals reviewed achieve these results. Although previous studies have compared different temporal algebras, ours is the first to evaluate a number of temporal algebras against a comprehensive set of well-defined, objective criteria.

9.2 Conclusions

This research has shown that the snapshot algebra can be extended to support query and update of temporal databases, while also accommodating the incremental update of materialized views. The algebraic language that we defined is sufficient to support the incremental update of materialized views in the context of general support for query and update of temporal databases. Also, our prototype implementation of an incremental query processor for TQuel serves as proof that implementation of the language is possible.

Definition of the language, in addition to proving our thesis, provided us insight to several issues central to the problem of extending the relational algebra to include support for valid time and transaction time. We present here our observations, some of which we recognize are controversial, concerning these issues.

- *A historical algebra should be consistent with the user-oriented conceptual visualization of historical relation states as three-dimensional objects.* This pervasive “spatial metaphor” [Ariav 1986, Ariav & Clifford 1986, Brooks 1956, Clifford & Tansel 1985] of a historical relation state provides a conceptual framework, at the users’ level, for assigning meaning to the database object that is a historical relation state. Hence, a historical algebra’s definition for a historical relation state should be consistent with this spatial metaphor. Furthermore, the algebra’s definition for each of its operators should be consistent with the conceptual visualization of an operation on historical relation states as a volume operation on spatial objects. Otherwise, the algebra will be inconsistent with user intuition for operations on historical relation states.
- *The treatment of valid time and transaction time cannot be uniform.* Transaction time has a specific semantics, very different from that of valid time, that requires special handling on update. Valid time is specified by the user and its value can be derived, via an algebraic expression, from values in underlying relations. Transaction time, however, is simply the time, as measured by a system clock, when update occurs. Its value can’t be specified by the user or derived from underlying relations. Although valid time and transaction time can be represented in a uniform manner [Ben-Zvi 1982, Gadia & Yeung 1988], there is no consistent interpretation for all query and update operations that accommodates a uniform treatment of the two aspects of time. We elected to associate transaction time with relation states to make formal definition of the language as straightforward as possible in the presence of rollback operations and schema evolution, but we could have associated transaction time with either tuples or attributes without changing the language’s semantics. After doing this research, however, we do not believe it possible to define historical operators with a consistent conceptual basis that manipulate transaction time in the same way they manipulate valid time. Likewise, we do not believe it possible to define meaningful update operations that treat valid time and transaction time similarly. Note, however, that our non-uniform handling of valid time and transaction time does not preclude the language’s use as the underlying query evaluation mechanism for queries posed

in terms of transaction time [Gadia & Yeung 1988]. Queries of this type can be supported by first converting transaction time to an explicit attribute via the rollback operator, as discussed in Section 4.3, and then treating that attribute the same as any other explicit, user-defined attribute in evaluating the expression (perhaps involving aggregates) that denotes the answer to the query. By converting transaction time to an explicit attribute, we are able to support queries over transaction time without having to introduce new operators that are inconsistent with the spatial metaphor of historical relation states as tree-dimensional spatial objects.

- *Integrity constraints should be modeled as restrictions on database update operations, not as restrictions on the algebraic manipulations of relation states.* Although integrity constraints will be an essential part of any temporal data model, our historical algebra, like the snapshot algebra, is defined independently of any consideration for integrity constraints. Although we have not addressed the issue of integrity constraints, our language would properly support integrity constraints as additional predicates in the definitions of commands, rather than as extensions of the historical operators. Only by not considering the issue of integrity constraints were we able to define a historical algebra whose operators all satisfy the closure property of algebras.
- *Definition of a historical algebra should include historical versions of all the basic snapshot operators.* Definition of a historical algebra with a consistent conceptual basis for each of its operators is a relatively simple task when only a few operators are considered; it is a much more difficult task when historical counterparts of all five basic snapshot operators, as well as new historical operators, are considered. Also, definition of historical versions for some subset of operators does not guarantee that compatible definitions exist for the other operators. We gained as much insight to the problem of adding valid time to the snapshot algebra from defining historical union, difference, and cartesian product as we did from defining historical selection, projection, and join. We also found, to our initial surprise, that historical difference, in particular, restricted our options for adding valid time to the snapshot algebra.
- *Design decisions and algebraic properties are interdependent.* There are a few basic design decisions (c.f., Section 3.1) that one must make to add valid time to the snapshot algebra, the choices one makes to these decisions being important factors in determining the properties of the resulting algebra. Likewise, for an algebra to have a certain property, appropriate choices must be made to these design decisions. We found that, unfortunately, not all desirable properties of historical algebras are compatible and that many subtle issues arise when attempting to define an algebra that has several desirable properties. There simply is no combination of choices to design decisions for which the resulting historical algebra has all possible desirable properties. Hence, the best that can be hoped for when defining a historical algebra is an algebra with a maximal subset of the most desirable properties.

9.3 Future Work

The research presented here, while it poses a solution to the problem of adding valid time and transaction time to the relational algebra, suggests additional work in several areas. These areas for future work include temporal data models, language extensions, additional evaluation criteria, implementation issues, and language completeness.

One area for future work is definition of a temporal data model. The relational model consists of three components: a set of objects, a set of operations, and a set of integrity rules [Codd 1981, Date 1986E]. A temporal data model also should have all three of these components. Our language addresses the issue of defining temporal objects and operations on temporal objects in the context of general support for query and update of temporal databases, but it does not address the related issue of temporal integrity rules. Although temporal integrity rules have been studied [Ariav 1986, Gadia & Yeung 1988, Navathe & Ahmed 1987], the role of temporal keys and functional dependencies in our language has yet to be investigated. Hence, to define a temporal data model based on our language, we need to extend our language to include support for temporal integrity rules.

Another area for future work is definition of an algebra for signatures, analogous to those for snapshot and historical relation states. In Chapter 4 we required that signature specifications in commands be a relation's current signature or a constant. To remove this restriction, we need to define an algebra for signature specification that would support signature changes dependent on both the current and past signatures of relations in the database. There also are a number of other language extensions that are possible. These include

- Extension of the language to accommodate deferred update of materialized views,
- Extension of the historical algebra to support both multi-dimensional time-stamps and periodicity,
- Introduction of algebraic operators that map between the domain of snapshot states and the domain of historical states directly, and
- Definition of non-incremental and incremental versions of the historical algebra that support non-first-normal-form relations.

The set of evaluation criteria presented in Chapter 8 is meant to be exhaustive. Although it includes the known desirable properties of temporal algebras, we anticipate that additional desirable properties of temporal algebras will be identified as more attention is given to the role of time in databases.

Our incremental query processor for TQuel is only a prototype. Its development gave us some insight to the problems we are likely to encounter in implementing the language, but there are many implementation issues that have yet to be explored. Our current prototype is composed of two components: a code generator and an interpreter. For performance, we need to replace these component with a compiler containing a query

optimizer. The prototype supports neither aggregates nor all the commands. We need to extend the prototype to support these. Also, there is a need to develop algorithms for accommodating dynamic time-stamps efficiently, reducing the search space of interval assignments at historical derivation nodes, and implementing aggregates efficiently. An area of related work would be evaluation of the effect the implementation techniques discussed in Chapter 7 have on the performance of TDBMS's that support incremental maintenance of materialized views. Our TQuel prototype could be used, perhaps after being extended to support a more complete set of TQuel statements, as a testbed for these performance studies. Performance studies, such as the comparison of different techniques for cacheing intermediate relation states between network activations, are likely to provide additional insight to when various techniques should be used to implement efficient update networks for historical views.

Finally, language completeness is another area for future work. One approach is to define a language and propose it as a standard; Codd proposed his snapshot algebra as the yardstick for snapshot completeness (i.e., supporting neither valid time nor transaction time). Several others have proposed notions of query completeness based on computability [Abiteboul & Vianu 1987, Chandra & Harel 1980], which, unfortunately, are incomparable. We feel that some variation on this latter approach is preferable and await a consensus to form against which we could measure our language for rollback completeness, historical completeness, and temporal completeness.

Bibliography

- [Abiteboul & Vianu 1985] Abiteboul, S. and V. Vianu. *Transactions and Integrity Constraints*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. 1985, pp. 193-204.
- [Abiteboul & Vianu 1986] Abiteboul, S. and V. Vianu. *Deciding Properties of Transactional Schemas*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. 1986, pp. 235-239.
- [Abiteboul & Vianu 1987] Abiteboul, S. and V. Vianu. *A Transaction Language Complete for Database Update and Specification*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. San Diego, CA: Mar. 1987.
- [Adiba & Lindsay 1980] Adiba, M.E. and B.G. Lindsay. *Database Snapshots*, in *Proceedings of the Conference on Very Large Databases*. Montreal, Canada: Oct. 1980, pp. 86-91.
- [Adiba & Bui Quang 1986] Adiba, M.E. and N. Bui Quang. *Historical Multi-media Databases*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kamabayashi. Kyoto, Japan: Aug. 1986, pp. 63-70.
- [Ahn 1986A] Ahn, I. *Performance Modeling and Access Methods for Temporal Database Management Systems*. PhD. Diss. Computer Science Department, University of North Carolina at Chapel Hill, July 1986.
- [Ahn 1986B] Ahn, I. *Towards an Implementation of Database Management Systems with Temporal Support*, in *Proceeding of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 374-381.
- [Ahn & Snodgrass 1986] Ahn, I. and R. Snodgrass. *Performance Evaluation of a Temporal Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. C. Zaniolo. Association for Computing Machinery. Washington, DC: May 1986, pp. 96-107.
- [Ahn & Snodgrass 1988] Ahn, I. and R. Snodgrass. *Performance Analysis of Temporal Queries (to appear)*. *Information Sciences*, (1988).

- [Aho et al. 1979] Aho, A.V., Y. Sagiv and J.D. Ullman. *Efficient Optimization of a Class of Relational Expressions*. *ACM Transactions on Database Systems*, 4, No. 4, Dec. 1979, pp. 435-454.
- [Aho et al. 1986] Aho, A.V., R. Sethi and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Reading, Massachusetts: Addison Wesley, 1986.
- [Allen & Hayes 1985] Allen, J.F. and P.J. Hayes. *A Common-Sense Theory of Time*, in *Proceedings of the International Joint Conference on Artificial Intelligence*. Los Angeles, CA: Aug. 1985, pp. 528-531.
- [Anderson 1982] Anderson, T.L. *Modeling Time at the Conceptual Level*, in *Proceedings of the International Conference on Databases: Improving Usability and Responsiveness*. Ed. P. Scheuermann. Jerusalem, Israel: Academic Press, June 1982, pp. 273-297.
- [Ariav 1984] Ariav, G. *Preserving the Time Dimension in Information Systems*. PhD. Diss. The Wharton School, University of Pennsylvania, Apr. 1984.
- [Ariav 1986] Ariav, G. *A Temporally Oriented Data Model*. *ACM Transactions on Database Systems*, 11, No. 4, Dec. 1986, pp. 499-527.
- [Ariav & Clifford 1986] Ariav, G. and J. Clifford. *Temporal Data Management: Models and Systems*, in *New Directions for Database Systems*. Norwood, New Jersey: Ablex Publishing Corporation, 1986. Chap. 12. pp. 168-185.
- [Bancilhon & Spyros 1981] Bancilhon, F. and N. Spyros. *Update Semantics of Relational Views*. *ACM Transactions on Database Systems*, 6, No. 4, Dec. 1981, pp. 557-575.
- [Banerjee et al. 1987] Banerjee, J., W. Kim, H.-J. Kim and H.F. Korth. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: 1987, pp. 311-322.
- [Ben-Zvi 1982] Ben-Zvi, J. *The Time Relational Model*. PhD. Diss. Computer Science Department, UCLA, 1982.
- [Bernstein et al. 1987] Bernstein, P.A., V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1987.
- [Blakeley et al. 1986A] Blakeley, J.A., P.-A. Larson and F.W. Tompa. *Efficiently Updating Materialized Views*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. C. Zaniolo. Association for Computing Machinery. Washington, DC: May 1986, pp. 61-71.
- [Blakeley et al. 1986B] Blakeley, J.A., N. Coburn and P.-A. Larson. *Updating Derived Rela-*

- tions: Detecting Irrelevant and Autonomously Computable Updates, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 457-466.
- [Bolour et al. 1982] Bolour, A., T.L. Anderson, L.J. Dekeyser and H.K.T. Wong. *The Role of Time in Information Processing: A Survey*. *SigArt Newsletter*, 80, Apr. 1982, pp. 28-48.
- [Bontempo 1983] Bontempo, C. J. *Feature Analysis of Query-By-Example*, in *Relational Database Systems*. New York: Springer-Verlag, 1983. pp. 409-433.
- [Brodie 1981] Brodie, M.L. *On Modelling Behavioral Semantics of Databases*, in *Proceedings of the Conference on Very Large Databases*. Ed. C. Zaniolo and C. Delobel. Cannes, France: Sep. 1981, pp. 32-42.
- [Brooks 1956] Brooks, F.P. *The Analytic Design of Automatic Data Processing Systems*. PhD. Diss. Harvard University, May 1956.
- [Bubenko 1977] Bubenko, J.A., Jr. *The Temporal Dimension in Information Modeling*, in *Architecture and Models in Data Base Management Systems*. The Netherlands: North-Holland Pub. Co., 1977. pp. 93-118.
- [Ceri et al. 1981] Ceri, S., G. Pelagatti and G. Bracchi. *Structured Methodology for Designing Static and Dynamic Aspects of Data Base Applications*. *Information Systems*, 6, No. 1 (1981), pp. 31-45.
- [Ceri & Gottlob 1985] Ceri, S. and G. Gottlob. *Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries*. *IEEE Transactions on Software Engineering*, SE-11, No. 4, Apr. 1985, pp. 324-345.
- [Chakravarthy & Minker 1986] Chakravarthy, U.S. and J. Minker. *Multiple Query Processing in Deductive Databases Using Query Graphs*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 384-391.
- [Chakravarthy & Minker 1982] Chakravarthy, U.S., Minker, J. *Processing Multiple Queries in Database Systems*. *Database Engineering*, 5, No. 3, Sep. 1982, pp. 38-44.
- [Chamberlin et al. 1975] Chamberlin, D.D., J.N. Gray and I.L. Traiger. *Views, Authorization, and Locking in a Relational Data Base System*, in *AFIPS Conference Proceedings*. AFIPS. Anaheim, CA: 1975, pp. 425-430.
- [Chandra & Harel 1980] Chandra, A.K. and D. Harel. *Computable Queries for Relational Data Bases*. *Journal of Computer and System Sciences*, 21, No. 2, Oct. 1980, pp. 156-178.
- [Chazelle 1985] Chazelle, B. *How to Search in History*. *Information and Control*, 64 (1985), pp. 77-99.

- [Clifford & Warren 1983] Clifford, J. and D.S. Warren. *Formal Semantics for Time in Databases*. *ACM Transactions on Database Systems*, 8, No. 2, June 1983, pp. 214-254.
- [Clifford & Tansel 1985] Clifford, J. and A.U. Tansel. *On an Algebra for Historical Relational Databases: Two Views*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 247-265.
- [Clifford & Croker 1987] Clifford, J. and A. Croker. *The Historical Relational Data Model (HRDM) and Algebra Based on Lifespans*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1987, pp. 528-537.
- [Codd 1970] Codd, E.F. *A Relational Model of Data for Large Shared Data Banks*. *Communications of the Association of Computing Machinery*, 13, No. 6, June 1970, pp. 377-387.
- [Codd 1972] Codd, E.F. *Relational Completeness of Data Base Sublanguages*, in *Data Base Systems*. Vol. 6 of Courant Computer Symposia Series. Englewood Cliffs, N.J.: Prentice Hall, 1972. pp. 65-98.
- [Codd 1981] Codd, E.F. *Data Models in Database Management*. *ACM SIGMOD Record*, 11, No. 2, Feb. 1981.
- [Cole 1986] Cole, R. *Searching and Storing Similar Lists*. *Journal of Algorithms*, 7, No. 2, June 1986, pp. 202-220.
- [Cosmadakis & Papadimitriou 1984] Cosmadakis, S.S. and C.H. Papadimitriou. *Updates of Relational Views*. *Journal of the Association of Computing Machinery*, 31, No. 4, Oct. 1984, pp. 742-760.
- [Dadam et al. 1984] Dadam, P., V. Lum and H.-D. Werner. *Integration of Time Versions into a Relational Database System*, in *Proceedings of the Conference on Very Large Databases*. Ed. U. Dayal, G. Schlageter and L.H. Seng. Singapore: Aug. 1984, pp. 509-522.
- [Date 1976] Date, C. J. *An Introduction to Database Systems*. Systems Programming Series. Reading, MA: Addison-Wesley Publishing Company, 1976.
- [Date 1986A] Date, C.J. *Some Relational Myths Exploded: An Examination of Some Popular Misconceptions Concerning Relational Database Management Systems*, in *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, 1986. Chap. 6. pp. 77-123.
- [Date 1986B] Date, C.J. *The Relational Model and Its Interpretation*, in *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, 1986. Chap. 8. pp. 143-150.

- [Date 1986C] Date, C.J. *Relational Database: An Overview*, in *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, 1986. Chap. 1. pp. 3-19.
- [Date 1986D] Date, C.J. *An Introduction to Database Systems*. Vol. I of Addison-Wesley Systems Programming Series. Reading, MA: Addison-Wesley Pub. Co., Inc., 1986.
- [Date 1986E] Date, C.J. *A Formal Definition of the Relational Model*, in *Relational Database: Selected Writings*. Reading, MA: Addison-Wesley, 1986. Chap. 7. pp. 125-141.
- [Dobkin & Munro 1980] Dobkin, D.P. and J.I. Munro. *Efficient Uses of the Past*, in *Proceedings of the Annual Symposium on Foundations of Computer Science*. IEEE Computer Society. Syracuse, NY: IEEE Computer Society Press, Oct. 1980, pp. 200-206.
- [Dobkin & Munro 1985] Dobkin, D.P. and J.I. Munro. *Efficient Uses of the Past*. *Journal of Algorithms*, 6, No. 4, Dec. 1985, pp. 455-465.
- [Enderton 1977] Enderton, H.B. *Elements of Set Theory*. New York, N.Y.: Academic Press, Inc., 1977.
- [Finkelstein 1982] Finkelstein, S. *Common Expression Analysis in Database Applications*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. M. Schkolnick. Association for Computing Machinery. Orlando, FL: June 1982. pp. 235-245.
- [Freitag & Goodman 1986] Freitag, J.C. and N. Goodman. *Translating Aggregate Queries into Iterative Programs*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 138-146.
- [Furtado et al. 1979] Furtado, A.L., K.C. Sevcik and C.S. dos Santos. *Premitting Updates Through Views of Data Bases*. *Information Systems*, 4, No. 4 (1979).
- [Furtado & Casanova 1985] Furtado, A.L. and M.A. Casanova. *Updating Relational Views*, in *Query Processing in Database Systems*. Springer-Verlag, 1985. pp. 127-142.
- [Gadia & Vaishnav 1985] Gadia, S.K. and J.H. Vaishnav. *A Query Language for a Homogeneous Temporal Database*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. Mar. 1985, pp. 51-56.
- [Gadia 1986] Gadia, S.K. *Toward a Multihomogeneous Model for a Temporal Database*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1986, pp. 390-397.
- [Gadia 1988] Gadia, S.K. *A Homogeneous Relational Model and Query Languages for Temporal Databases (to appear)*. *ACM Transactions on Database Systems*, Dec. 1988.
- [Gadia & Yeung 1988] Gadia, S.K. and C.S. Yeung. *A Generalized Model for a Relational Temporal Database*, in *Proceedings of ACM SIGMOD International Conference on*

- Management of Data.* Association for Computing Machinery. Chicago, IL: June 1988, pp. 251-259.
- [Gerritsen & Morgan 1976] Gerritsen, R. and H.L. Morgan. *Dynamic Restructuring of Databases with Generation Data Structures*, in *Proceedings of the ACM Annual Conference*. Association for Computing Machinery. Houston, TX: Oct. 1976, pp. 281-286.
- [Gordon 1979] Gordon, M.J.C. *The Denotational Description of Programming Languages*. New York-Heidelberg-Berlin: Springer-Verlag, 1979.
- [Hall 1976] Hall, P.A.V. *Optimization of Single Expressions in a Relational Data Base System*. *IBM Journal of Research and Development*, 20, No. 3, May 1976, pp. 244-257.
- [Hammer & McLeod 1981] Hammer, M. and D. McLeod. *Database Description with SDM: A Semantic Database Model*. *ACM Transactions on Database Systems*, 6, No. 3, Sep. 1981, pp. 351-386.
- [Hanson 1987A] Hanson, E.N. *A Performance Analysis of View Materialization Strategies*, in *Proceedings of the ACM SIGMOD Annual Conference*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: ACM Press, May 1987, pp. 440-453.
- [Hanson 1987B] Hanson, E.N. *Efficient Support for Rules and Derived Objects in Relational Database Systems*. PhD. Diss. Computer Science Department, University of California at Berkeley, Aug. 1987.
- [Hanson 1988] Hanson, E.N. *Processing Queries Against Database Procedures: A Performance Analysis*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery. Chicago, IL: June 1988, pp. 295-302.
- [Held et al. 1975] Held, G.D., M. Stonebraker and J. Wong. *INGRES-A Relational Data Base Management System*, in *Proceedings of the AFIPS National Computer Conference*. Anaheim, CA: AFIPS Press, May 1975, pp. 409-416.
- [Horwitz 1985] Horwitz, S.B. *Generating Language-Based Editors: A Relationally-Attributed Approach*. PhD. Diss. Department of Computer Science, Cornell University, Aug. 1985.
- [Horwitz 1986] Horwitz, S.B. *Adding Relational Databases to Existing Software Systems: Implicit Relations and a New Relational Query Evaluation Method*. Technical Report 674. Computer Sciences Department, University of Wisconsin. Nov. 1986.
- [Horwitz & Teitelbaum 1986] Horwitz, S.B. and T. Teitelbaum. *Generating Editing Environments Based on Relations and Attributes*. *ACM Transactions on Programming Languages and Systems*, 8, No. 4, Oct. 1986, pp. 577-608.

- [IBM 1981] IBM SQL/Data-System, Concepts and Facilities. Technical Report GH24-5013-0. IBM. Jan. 1981.
- [Jarke & Koch 1984] Jarke, M. and J. Koch. *Query Optimization in Database Systems*. ACM Computing Surveys, 16, No. 2, June 1984, pp. 111-152.
- [Jones et al. 1979] Jones, S., P. Mason and R. Stamper. *LEGOL 2.0: A Relational Specification Language for Complex Rules*. Information Systems, 4, No. 4, Nov. 1979, pp. 293-305.
- [Kahler & Risnes 1987] Kahler, B and O. Risnes. *Extending Logging for Database Snapshot Refresh*, in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 389-398.
- [Keller 1985] Keller, A.M. *Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. Association for Computing Machinery. Mar. 1985.
- [Keller 1986] Keller, A.M. *Choosing a View Update Translator by Dialog at View Definition Time*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 467-474.
- [Klopprogge 1981] Klopprogge, M.R. *TERM: An Approach to Include the Time Dimension in the Entity-Relationship Model*, in *Proceedings of the Second International Conference on the Entity Relationship Approach*. Washington, DC: Oct. 1981, pp. 477-512.
- [Klug 1982] Klug, A. *Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions*. Journal of the Association of Computing Machinery, 29, No. 3, July 1982, pp. 699-717.
- [Koenig & Paige 1981] Koenig, S. and R. Paige. *A Transformational Framework for the Automatic Control of Derived Data*, in *Proceedings of the Conference on Very Large Databases*. Ed. C. Zaniolo and C. Delobel. Cannes, France: Sep. 1981, pp. 306-318.
- [Krishnamurthy et al. 1986] Krishnamurthy, R., H. Boral and C. Zaniolo. *Optimization of Nonrecursive Queries*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 128-137.
- [Larson & Yang 1985] Larson, F.-A. and H.Z. Yang. *Computing Queries from Derived Relations*, in *Proceedings of the Conference on Very Large Databases*. Ed. A. Pirotte and Y. Vassiliou. Stockholm, Sweden: Aug. 1985, pp. 259-269.
- [Lee 1985] Lee, R.M. *A Denotational Semantics for Administrative Databases*, in *Proceedings of the IFIP WG 2.6 Working Conference on Data Semantics (DS-1)*. Ed. T.B. Steel and R. Meersman. IFIP. Hasselt, Belgium: Jan. 1985, pp. 83-120.

- [Lehman & Carey 1986] Lehman, T.J. and M.J. Carey. *A Study of Index Structures for Main Memory Database Management Systems*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 294-303.
- [Lindsay et al. 1986] Lindsay, B., L. Haas, C. Mohan, H. Pirahesh and P. Wilms. *A Snapshot Differential Refresh Algorithm*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. C. Zaniolo. Association for Computing Machinery. Washington, DC: May 1986, pp. 53-60.
- [Lorentzos & Johnson 1987A] Lorentzos, N.A. and R.G. Johnson. *TRA: A Model for a Temporal Relational Algebra*, in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 99-112.
- [Lorentzos & Johnson 1987B] Lorentzos, N.A. and R.G. Johnson. *Extending Relational Algebra to Manipulate Temporal Data*. Internal Report NL/1/87. Department of Computer Science, Birkbeck College, London University. Aug. 1987.
- [Lum et al. 1984] Lum, V., P. Dadam, R. Erbe, J. Guenauer, P. Pistor, G. Walch, H. Werner and J. Woodfill. *Designing DBMS Support for the Temporal Dimension*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. B Yormark. Association for Computing Machinery. Boston, MA: June 1984, pp. 115-130.
- [Maier 1983] Maier, D. *The Theory of Relational Databases*. Rockville, MD: Computer Science Press, 1983.
- [Manola & Dayal 1986] Manola, F. and U. Dayal. *PDM: An Object-Oriented Data Model*, in *Proceedings of the International Workshop on Object-Oriented Database Systems*. 1986.
- [Markowitz & Makowsky 1987] Markowitz, V.M. and J.A. Makowsky. *Incremental Reorganization of Relational Databases*, in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 127-135.
- [Martin et al. 1987] Martin, N.G., S.B. Navathe and R. Ahmed. *Dealing with Temporal Schema Anomalies in History Databases*, in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 177-184.
- [McKenzie 1986] McKenzie, E. *Bibliography: Temporal Databases*. *ACM SIGMOD Record*, 15, No. 4, Dec. 1986, pp. 40-52.
- [Myers 1984] Myers, E.W. *Efficient Applicative Data Types*, in *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*. Salt Lake City, UT: Jan. 1984, pp. 66-75.
- [Navathe & Fry 1976] Navathe, S.B. and J.P. Fry. *Restructuring for Large Databases: Three Levels of Abstraction*. *ACM Transactions on Database Systems*, 1, no. 2, June 1976, pp. 138-158.

- [Navathe 1980] Navathe, S.B. *Schema Analysis for Database Restructuring*. *ACM Transactions on Database Systems*, 5, No. 2, June 1980, pp. 157-184.
- [Navathe & Ahmed 1986] Navathe, S.B. and R. Ahmed. *A Temporal Relational Model and a Query Language*. UF-CIS Technical Report TR-85-16. Computer and Information Sciences Department, University of Florida. Apr. 1986.
- [Navathe & Ahmed 1987] Navathe, S.B. and R. Ahmed. *TSQL-A Language Interface for History Databases*, in *Proceedings of the Conference on Temporal Aspects in Information Systems*. AFCET. France: May 1987, pp. 113-128.
- [Nievergelt et al. 1984] Nievergelt, J., H. Hinterberger and K. C. Sevcik. *The Grid File: An Adaptable, Symmetric Multikey File Structure*. *ACM Transactions on Database Systems*, 9, No. 1, Mar. 1984, pp. 38-71.
- [Overmars 1981A] Overmars, M.H. *Searching in the Past, I*. Technical Report RUU-CS-81-7. Rijksuniversiteit Utrecht. Apr. 1981.
- [Overmars 1981B] Overmars, M.H. *Searching in the Past, II: General Transforms*. Technical Report RUU-CS-81-9. Rijksuniversiteit Utrecht. May 1981.
- [Overmars 1983] Overmars, M.H. *The Design of Dynamic Data Structures*. PhD. Diss. Rijksuniversiteit Utrecht, 1983.
- [Overmyer & Stonebraker 1982] Overmyer, R. and M. Stonebraker. *Implementation of a Time Expert in a Database System*. *ACM SIGMOD Record*, 12, No. 3, Apr. 1982, pp. 51-59.
- [Ozsoyoglu et al. 1987] Ozsoyoglu, G., Z. Ozsoyoglu and V. Matos. *Extending Relational Algebra and Relational Calculus with Set-Valued Attributes and Aggregate Functions*. *ACM Transactions on Database Systems*, 12, No. 4, Dec. 1987, pp. 566-592.
- [Rishe 1985] Rishe, N. *On Denotational Semantics of Data Bases*, in *Proceedings of the International Conference on Mathematical Foundations of Programming Semantics*. Ed. A. Melton. Manhattan, KA: Springer-Verlag, Apr. 1985, pp. 249-274.
- [Rotem & Segev 1987] Rotem, D. and A. Segev. *Physical Organization of Temporal Databases*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1987, pp. 547-553.
- [Roth et al. 1984] Roth, M.A., H.F. Korth and A. Silberschatz. *Extended Algebra and Calculus for Non-1NF Relational Databases*. Technical Report TR-84-36. Department of Computer Sciences, University of Texas at Austin. Dec. 1984.
- [Roussopoulos 1982A] Roussopoulos, N. *The Logical Access Path Schema of a Database*. *IEEE Transactions on Software Engineering*, SE-8, No. 6, Nov. 1982, pp. 563-573.
- [Roussopoulos 1982B] Roussopoulos, N. *View Indexing in Relational Databases*. *ACM*

- Transactions on Database Systems*, 7, No. 2, June 1982, pp. 258-290.
- [Roussopoulos & Yeh 1984] Roussopoulos, N. and R.T. Yeh. *An Adaptable Methodology for Database Design*. Computer, , May 1984, pp. 64-80.
- [Roussopoulos & Mark 1985] Roussopoulos, N. and L. Mark. *Schema Manipulation in Self-Describing and Self-Documenting Data Models*. International Journal of Computer and Information Sciences, 14, No. 1, Jan. 1985, pp. 1-28.
- [Roussopoulos & Kang 1986A] Roussopoulos, N. and H. Kang. *Principles and Techniques in the Design of ADMS*. Computer, 19, No. 12, Dec. 1986, pp. 19-25.
- [Roussopoulos & Kang 1986B] Roussopoulos, N. and H. Kang. *Preliminary Design of ADMS: A Workstation-Mainframe Integrated Architecture for Database Management Systems*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 355-364.
- [Roussopoulos 1987] Roussopoulos, N. *Overview of ADMS: A High Performance Database Management System*. Technical Report. University of Maryland. 1987.
- [Sadeghi 1987] Sadeghi, R. *A Database Query Language for Operations on Historical Data*. PhD. Diss. Dundee College of Technology, Dec. 1987.
- [Sarda 1988] Sarda, N.L. *Algebra and Query Language for a Historical Data Model*. The Computer Journal, (1988).
- [Sarnak & Tarjan 1986] Sarnak, N. and R.E. Tarjan. *Planar Point Location Using Persistent Search Trees*. Communications of the ACM, 29, No. 7, July 1986, pp. 669-679.
- [Satoh et al. 1985] Satoh, K., M. Tsuchida, F. Nakamura and K. Oomachi. *Local and Global Query Optimization Mechanisms for Relational Databases*, in *Proceedings of the Conference on Very Large Databases*. Ed. A. Pirotte and Y. Vassiliou. Stockholm, Sweden: Aug. 1985, pp. 405-417.
- [Schek & Scholl 1986] Schek, H.-J., Scholl, M.H. *The Relational Model with Relation-valued Attributes*. Information Systems, 11, No. 2 (1986), pp. 137-147.
- [Schmidt 1986] Schmidt, D.A. *Denotational Semantics, A Methodology for Language Development*. Newton, Massachusetts: Allyn and Bacon, 1986.
- [Scott 1976] Scott, D.S. *Data Types as Lattices*. SIAM Journal of Computing, 5, No. 3, Sep. 1976, pp. 522-587.
- [Segev & Shoshani 1987] Segev, A. and A. Shoshani. *Logical Modeling of Temporal Data*, in *Proceedings of the ACM SIGMOD Annual Conference on Management of Data*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: ACM Press, May 1987, pp. 454-466.

- [Selinger et al. 1979] Selinger, P.G., M.M. Astrahan, D.D. Chamberlin, R.A. Lorie and T.G. Price. *Access Path Selection in a Relational Database Management System*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. P.A. Bernstein. Association for Computing Machinery. Boston, MA: 1979, pp. 23-34.
- [Sellis & Shapiro 1985] Sellis, T.K. and L. Shapiro. *Optimization Of Extended Database Query Languages*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 424-436.
- [Severance & Lohman 1976] Severance, D.G. and G.M. Lohman. *Differential Files: Their Application to the Maintenance of Large Databases*. *ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 256-267.
- [Shiftan 1986] Shiftan, J. *An Assessment of the Temporal Differentiation of Attributes in the Implementation of a Temporally Oriented DBMS*. PhD. Diss. Information Systems Area, Graduate School of Business Administration, New York University, Aug. 1986.
- [Shmueli & Atai 1984] Shmueli, O. and A. Itai. *Maintenance of Views*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. B. Yormark. Association for Computing Machinery. Boston, MA: ACM Press, June 1984, pp. 240-255.
- [Shmueli & Itai 1987] Shmueli, O. and A. Itai. *Complexity of Views: Tree and Cyclic Schemas*. *SIAM Journal on Computing*, 16, No. 1, Feb. 1987, pp. 17-37.
- [Shoshani & Kawagoe 1986] Shoshani, A. and K. Kawagoe. *Temporal Data Management*, in *Proceedings of the Conference on Very Large Databases*. Ed. Y. Kambayashi. Kyoto, Japan: Aug. 1986, pp. 79-88.
- [Shu et al. 1977] Shu, N.C., B.C. Housel, R.W. Taylor, S.P. Ghosh and V.Y. Lum. *EX-PRESS: A Data EXtraction, Processing, and REstructuring System*. *ACM Transactions on Database Systems*, 2, No. 2, June 1977, pp. 134-174.
- [Shu 1987] Shu, N.C. *Automatic Data Transformation and Restructuring*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1987, pp. 173-180.
- [Smith & Chang 1975] Smith, J.M. and P.Y-T. Chang. *Optimizing the Performance of a Relational Algebra Database Interface*. *Communications of the Association of Computing Machinery*, 18, No. 10, Oct. 1975, pp. 568-579.
- [Snodgrass 1982] Snodgrass, R. *Monitoring Distributed Systems: A Relational Approach*. PhD. Diss. Computer Science Department, Carnegie-Mellon University, Dec. 1982.
- [Snodgrass & Ahn 1985] Snodgrass, R. and I. Ahn. *A Taxonomy of Time in Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*.

- Ed. S. Navathe. Association for Computing Machinery. Austin, TX: May 1985, pp. 236-246.
- [Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases*. IEEE Computer, 19, No. 9, Sep. 1986, pp. 35-42.
- [Snodgrass 1987] Snodgrass, R. *The Temporal Query Language TQuel*. ACM Transactions on Database Systems, 12, No. 2, June 1987, pp. 247-298.
- [Snodgrass et al. 1987] Snodgrass, R., S. Gomez and E. McKenzie. *Aggregates in the Temporal Query Language TQuel*. TempIS Technical Report 16. Computer Science Department, University of North Carolina at Chapel Hill. July 1987.
- [Snodgrass 1988] Snodgrass, R. *The Interface Description Language: Definition and Use (forthcoming)*. Rockville, MD: Computer Science Press, 1988.
- [Sockut & Goldberg 1979] Sockut, G.H. and R.P. Goldberg. *Database Reorganization - Principles and Practice*. ACM Computing Surveys, 11, No. 4, Dec. 1979, pp. 371-395.
- [Stam & Snodgrass 1988] Stam, R. and R. Snodgrass. *A Bibliography of Temporal Databases*. Database Engineering, 7, No. 4, Dec. 1988.
- [Stonebraker 1975] Stonebraker, M. *Implementation of Integrity Constraints and Views by Query Modification*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery. San Jose, CA: June 1975.
- [Stonebraker et al. 1976] Stonebraker, M., E. Wong, P. Kreps and G. Held. *The Design and Implementation of INGRES*. ACM Transactions on Database Systems, 1, No. 3, Sep. 1976, pp. 189-222.
- [Stonebraker 1987] Stonebraker, M. *The Design of the POSTGRES Storage System*, in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 289-300.
- [Stoy 1977] Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Series in Computer Science. The MIT Press, 1977.
- [Strachey 1966] Strachey, C. *Towards a Formal Semantics*, in *Formal Language Description Languages for Computer Programming*. North Holland, 1966. pp. 198-220.
- [Subieta 1987] Subieta, K. *Denotational Semantics of Query Languages*. Information Systems, 12, No. 1 (1987), pp. 69-82.
- [Tandem 1983] Tandem Computers, Inc. *ENFORM Reference Manual*. Cupertino, CA, 1983.

- [Tansel & Arkun 1985] Tansel, A.U. and M.E. Arkun. *Equivalence of Historical Relational Calculus and Historical Relational Algebra*. Technical Report. Bernard M. Baruch College, CUNY. 1985.
- [Tansel et al. 1985] Tansel, A.U., M.E. Arkun and G. Ozsoyoglu. *Time-By-Example Query Language for Historical Databases*. Technical Report. Bernard M. Baruch College, CUNY. 1985.
- [Tansel 1986] Tansel, A.U. *Adding Time Dimension to Relational Model and Extending Relational Algebra*. *Information Systems*, 11, No. 4 (1986), pp. 343-355.
- [Tansel & Arkun 1986] Tansel, A.U. and M.E. Arkun. *HQuel, A Query Language for Historical Relational Databases*, in *Proceedings of the Third International Workshop on Statistical and Scientific Databases*. July 1986.
- [Tansel 1987] Tansel, A.U. *A Statistical Interface for Historical Relational Databases*, in *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society. Los Angeles, CA: IEEE Computer Society Press, Feb. 1987, pp. 538-546.
- [Thirumalai & Krishna 1988] Thirumalai, S. and S. Krishna. *Data Organization for Temporal Databases*. Technical Report. Raman Research Institute, India. 1988.
- [Ullman 1982] Ullman, J.D. *Principles of Database Systems, Second Edition*. Potomac, Maryland: Computer Science Press, 1982.
- [Verma & Lu 1987] Verma, V. and H. Lu. *A New Approach to Version Management for Databases*, in *Proceedings of the AFIPS National Computer Conference*. Chicago, IL: AFIPS Press, June 1987, pp. 645-651.
- [Vianu 1983] Vianu, V. *Dynamic Constraints and Database Evolution*, in *Proceedings of the ACM Symposium on Principles of Database Systems*. Association for Computing Machinery. Atlanta, GA: Mar. 1983, pp. 389-399.
- [Woelk et al. 1986] Woelk, D., W. Kim and W. Luther. *An Object-Oriented Approach to Multimedia Databases*, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. C. Zaniolo. Washington, DC: Association for Computing Machinery, May 1986, pp. 311-325.
- [Wong & Youssefi 1976] Wong, E. and K. Youssefi. *Decomposition - A Strategy for Query Processing*. *ACM Transactions on Database Systems*, 1, No. 3, Sep. 1976, pp. 223-241.
- [Woodfill & Stonebraker 1983] Woodfill, J. and M. Stonebraker. *An Implementation of Hypothetical Relations*, in *Proceedings of the Conference on Very Large Databases*. Ed. M. Schkolnick and C. Thanos. Florence, Italy: 1983, pp. 157-166.
- [Yang & Larson 1987] Yang, H.Z. and P.-A. Larson. *Query Transformation for PSJ-queries*, in *Proceedings of the Conference on Very Large Databases*. Ed. P. Hammersley. Brighton, England: Sep. 1987, pp. 245-254.

[Yao 1979] Yao, S.B. *Optimization of Query Evaluation Algorithms. ACM Transactions on Database Systems*, 4, No. 2, June 1979, pp. 133-155.

[Yeung 1986] Yeung, C.S. *Query Languages for a Heterogeneous Temporal Database*. Master's Thesis, EE/CS Department, Texas Tech University, 1986.

Appendix A

Symbols

This appendix describes briefly the symbols used in the main body of the paper. It also identifies the page where each symbol is either defined or first used.

| <i>Symbols</i> | <i>Usage</i> | <i>Page</i> |
|----------------------------|--|-------------|
| \times | Conventional cartesian product operator | 68 |
| $\hat{\times}$ | Historical cartesian product operator | 28 |
| $\times^1, \hat{\times}^1$ | Incremental cartesian product operators | 143, 152 |
| δ | Historical derivation operator | 34 |
| δ^1 | Incremental historical derivation operator | 148 |
| $-$ | Conventional difference operator, set difference | 27 |
| \vdash | Historical difference operator | 27 |
| $-^1, \vdash^1$ | Incremental difference operators | 142, 151 |
| Δ | Differential | 138 |
| \div | Conventional division operator | 48 |
| $\hat{\div}$ | Historical division operator | 48 |
| \cap | Conventional intersection operator, set intersection | 46 |
| $\hat{\cap}$ | Historical intersection operator | 46 |
| \bowtie | Conventional natural join operator | 47 |
| \bowtie^1 | Historical natural join operator | 47 |
| π | Conventional projection operator | 68 |
| $\hat{\pi}$ | Historical projection operator | 30 |
| $\pi^1, \hat{\pi}^1$ | Incremental projection operators | 141, 149 |

| | | |
|---------------------------------------|---|----------|
| Θ | Conventional Θ -join | 46 |
| $\hat{\Theta}$ | Historical Θ -join | 47 |
| ρ | Rollback operator | 57 |
| $\hat{\rho}$ | Historical rollback operator | 57 |
| σ | Conventional selection operator | 68 |
| $\hat{\sigma}$ | Historical selection operator | 29 |
| $\sigma^I, \hat{\sigma}^I$ | Incremental selection operators | 141, 148 |
| \cup | Conventional union operator, set union | 26 |
| $\hat{\cup}$ | Historical union operator | 26 |
| $\cup^I, \hat{\cup}^I$ | Incremental union operators | 142, 150 |
| α, χ, ν | Temporal expressions, syntactic form | 106 |
| $\Phi_\alpha, \Phi_\chi, \Phi_\nu$ | Temporal expressions, semantic form | 107 |
| α', χ', ν' | TQuel temporal expressions, syntactic form | 104 |
| $\Phi'_\alpha, \Phi'_\chi, \Phi'_\nu$ | TQuel temporal expressions, semantic form | 105 |
| ψ | Boolean predicate, syntactic form | 106 |
| Ψ_ψ | Boolean predicate, semantic form | 107 |
| ψ' | TQuel boolean predicate, syntactic form | 104 |
| Ψ'_ψ | TQuel predicate expression, semantic form | 105 |
| τ | Temporal predicate, syntactic form | 106 |
| Γ_τ | Temporal predicate, semantic form | 107 |
| τ' | TQuel temporal predicate, syntactic form | 104 |
| Γ'_τ | TQuel temporal predicate, semantic form | 105 |
| A, A_u | Set of attributes induced by a relation signature | 24 |
| \hat{A} | Historical aggregation function for non-unique aggregates | 40 |
| \widehat{AU} | Historical aggregation function for unique aggregates | 43 |
| $\hat{A}^I, \widehat{AU}^I$ | Incremental historical aggregation functions | 153 |
| a, b, c | Attribute variables | 30 |
| B | By list | 37 |
| C, C_u | Command, syntactic form | 57 |

| | | |
|-----------------------------|---|-----|
| \mathcal{D} | Domain of value domains | 60 |
| \mathcal{D}_u | Arbitrary value domain | 23 |
| d | Database state | 66 |
| E, E_u | Expression, syntactic form | 57 |
| e | Number of value domains | 23 |
| F | Predicate in the selection operators | 28 |
| f | Scalar aggregate function | 40 |
| G | Predicate in the historical derivation operator | 32 |
| g, j | Relation variables | 105 |
| $g, J,$ | Distinct relation variables in aggregates | 118 |
| \mathcal{H}_z | Domain of historical relation states for signature z | 24 |
| H | Historical state, syntactic form | 57 |
| h, l | Variables ranging over attributes in target list, by-list, or aggregate | 30 |
| ht, ht' | Historical tuples | 24 |
| $I, I', I_u, I_{u,v}$ | Identifier | 24 |
| i | Relation variable | 104 |
| \mathcal{IN} | Domain of intervals | 299 |
| $\mathcal{P}(\mathcal{IN})$ | Power set of \mathcal{IN} | 299 |
| IN, IN_u | Interval | 34 |
| k | Number of relations | 103 |
| M | Relation's $MSoT$ | 80 |
| m, m_u | Number of attributes induced by a relation's signature | 26 |
| N | Decimal numeral | 57 |
| n | Length of target list or by-list | 29 |
| P, P_u | Program, syntactic form | 56 |
| p, y | Number of attributes appearing in an aggregate | 118 |
| p, \bar{y} | Number of distinct attributes appearing in an aggregate | 118 |
| Q, R, R_u | Historical relation states | 26 |
| q, r, r_u | Historical tuple variables | 26 |
| Q', R', R'_u | TQuel relations | 101 |

| | | |
|--------------------|---|-----|
| q', r', r'_u | TQuel tuple variables | 101 |
| S | Snapshot state, syntactic form | 57 |
| st, st' | Snapshot tuple | 60 |
| \mathcal{T} | Time domain | 24 |
| $\wp(\mathcal{T})$ | Power set of \mathcal{T} | 24 |
| T | Subset of \mathcal{T} | 277 |
| t, t_u | Element of \mathcal{T} | 24 |
| tn | Transaction number | 66 |
| U, u, v, x | Temporary variables | 23 |
| V, V_u | Temporal function in the historical derivation operator | 32 |
| W | Aggregation window function, syntactic form | 57 |
| w | Aggregation window function | 37 |
| X | Set of attributes names | 29 |
| Y | Relation class, syntactic form | 57 |
| Z | Relation signature, syntactic form | 57 |
| z, z_u | Relation signature | 23 |

Appendix B

Auxiliary Functions

In this appendix we present formal definitions for the auxiliary functions that appear in Chapters 3 and 4.

B.1 Semantic Functions

Several auxiliary semantic functions appear in the definitions of the semantic functions for expressions and commands in Chapters 4 and refCHViews. We present here formal definitions for each of those auxiliary semantic functions, along with formal definitions for all semantic functions used, in turn, in their definitions. For these definitions, we assume that we are given

- The value domains $\mathcal{D}_1, \dots, \mathcal{D}_e$;
- The semantic functions D_1, \dots, D_e , where D_x , $1 \leq x \leq e$, maps each string in the syntactic category *STRING* onto either an element of \mathcal{D}_x or *ERROR*; and
- A semantic function *DN* that maps identifiers in the syntactic category *IDENTIFIER* that denote a value domain (i.e., name a value domain) onto that domain and all other identifiers onto *UNBOUND*.
- A semantic function *WN* that maps identifiers in the syntactic category *IDENTIFIER* that denote an aggregation windowing function onto that function and all other identifiers onto *UNBOUND*.

For these definitions, let

B range over the category *BY LIST*;

F, F_1 , and F_2 range over the category *SIGMA EXPRESSION*;

FT range over the category *SIGMA TERM*;

G, G_1 , and G_2 range over the category *DELTA EXPRESSION*;

*GF*₁ and *GF*₂ range over the category *DELTA FACTOR*;

GT range over the category Δ -TERM;
 HT , HT_1 , and HT_2 range over the category H -TUPLE;
 I , I' , I_1 , I_2 , $I_{1,1}$, $I_{1,2}$, $I_{2,1}$, and $I_{2,2}$ range over the category IDENTIFIER;
 RO range over the category REL OP;
 SO range over the category SET OP;
 S , S_1 , and S_2 range over the category STRING;
 ST , ST_1 , and ST_2 range over the category S-TUPLE;
 T , T_1 , and T_2 range over the category TIME CONSTANT;
 TS , TS_1 , and TS_2 range over the category TIME SET;
 V , V_1 , and V_2 range over the category TIME EXPRESSION;
 h range over the domain HISTORICAL STATE;
 ht and ht' range over the domain HISTORICAL TUPLE;
 s range over the domain SNAPSHOT STATE;
 st and st' range over the domain SNAPSHOT TUPLE;
 u range over the domain $[\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times [\text{TRANSACTION NUMBER} + \{-\}]^*]$;
 v range over the domain $[\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^*$;
 w range over the domain $[\text{SNAPSHOT STATE} \times \text{TRANSACTION NUMBER}]^+ \times [\text{HISTORICAL STATE} \times \text{TRANSACTION NUMBER}]^*$;
 z range over the domain RELATION SIGNATURE.

Unfortunately, some of these conflict with the usage as given in Appendix A; such conflict was unavoidable. Also, unless specified otherwise, function definitions that involve the semantic domain RELATION assume the definition of RELATION given on page 61.

B is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category BY LIST onto an element in $\wp(\text{IDENTIFIER})$, the power set of IDENTIFIER, if the identifiers denote a valid subset of the attributes in a given signature. Otherwise, B maps the list onto ERROR.

$$B : [BY \ LIST \rightarrow SIGNATURE] \rightarrow [\wp(\text{IDENTIFIER}) + \{\text{ERROR}\}]$$

$$B[(\)]z = \emptyset$$

```

B[(I)] z = if z(I) = Dx then {I} else ERROR

B[(I1, I2...)] z =
  if (z(I1) = Dx
      ∧ B[(I2...)](z - {(I1, Dx)}) ∪ {(I1, UNBOUND)}) ≠ ERROR)
  then {I} ∪ B[(I2...)](z - {(I1, Dx)}) ∪ {(I1, UNBOUND)})
  else ERROR

```

F is a semantic function that maps the alphanumeric representation of a boolean predicate in the syntactic category *SIGMA EXPRESSION* onto its corresponding boolean predicate in the semantic domain *SELECTION PREDICATE*, if it denotes a valid boolean predicate for the selection operator σ (or $\hat{\sigma}$) and a given signature. Otherwise, **F** maps the predicate onto **ERROR**.

F : [*SIGMA EXPRESSION* → *SIGNATURE*] –
 [*SELECTION PREDICATE* + { **ERROR** }]

```

F[FT] z = FT[FT] z

F[F1 and F2] z =
  if (VALIDF[F1] z ∧ VALIDF[F2] z)
  then F[F1] z ∧ F[F2] z
  else ERROR

F[F1 or F2] z =
  if (VALIDF[F1] z ∧ VALIDF[F2] z)
  then F[F1] z ∨ F[F2] z
  else ERROR

F[not F] z = if VALIDF[F] z then ¬F[F] z else ERROR

F[(F)] z = (F[F] z)

```

FT is a semantic function that maps the alphanumeric representation of a term in the syntactic category *SIGMA TERM* onto its corresponding term in the semantic domain *SELECTION TERM*, if it denotes a valid term in a boolean predicate for the selection operator σ (or $\hat{\sigma}$) and a given signature. Otherwise, **FT** maps the term onto **ERROR**.

FT : [*SIGMA TERM* → *SIGNATURE*] → [*SELECTION TERM* + { **ERROR** }]

```

FT[I1 RO I2]z = if VALIDFT[I1 RO I2]z then I1 RO[RO] I2 else ERROR

FT[I RO S]z =
    if      (z(I) =  $D_x$   $\wedge$  VALIDFT[I RO S]z)
    then   I RO[RO]  $D_x[S]$ 
    else   ERROR

FT[S RO I]z =
    if      (z(I) =  $D_x$   $\wedge$  VALIDFT[S RO I]z)
    then    $D_x[S]$  RO[RO] I
    else   ERROR

```

G is a semantic function that maps the alphanumeric representation of a temporal predicate in the syntactic category *DELTA EXPRESSION* onto its corresponding temporal predicate in the semantic domain *DERIVATION PREDICATE*, if it denotes a valid temporal predicate for the derivation operator δ and a given signature. Otherwise, **G** maps the predicate onto **ERROR**.

G : [*DELT A EXPRESSION* → *SIGNATURE*] →
[*DERIVATION PREDICATE* + { *ERROR* }]

G[true] $z = \text{TRUE}$
G[GT] $z = \text{GT}[G_1, G_2] z$
G[G₁ and G₂] $z =$
 if (**VALIDG[G₁]** $z \wedge \text{VALIDG}[G_2] z$)
 then **G[G₁]** $z \wedge \text{G}[G_2] z$
 else **ERROR**
G[G₁ or G₂] $z =$
 if (**VALIDG[G₁]** $z \wedge \text{VALIDG}[G_2] z$)
 then **G[G₁]** $z \vee \text{G}[G_2] z$
 else **ERROR**

$G[\text{not } G]z = \text{if } \text{VALIDG}[G]z \text{ then } \neg G[G]z \text{ else } \text{ERROR}$

$G[(G)]z = \text{if } \text{VALIDG}[G]z \text{ then } (G[G]z) \text{ else } \text{ERROR}$

GF is a semantic function that maps the alphanumeric representation of a factor in the syntactic category DELT A FACTOR onto its corresponding factor in the semantic domain DERIVATION FACTOR , if it denotes a valid factor in a boolean predicate for the derivation operator δ and a given signature. Otherwise, GF maps the factor onto ERROR .

$\text{GF} : [\text{DELT A FACTOR} \rightarrow \text{SIGNATURE}] \rightarrow$

$[\text{DERIVATION FACTOR} + \{\text{ERROR}\}]$

$\text{GF}[T]z = N[T]$

$\text{GF}[\text{FIRST}(V)]z = \text{if } \text{VALIDTE}[V]z \text{ then } \text{First}(\text{TE}[V]z) \text{ else } \text{ERROR}$

$\text{GF}[\text{LAST}(V)]z = \text{if } \text{VALIDTE}[V]z \text{ then } \text{Last}(\text{TE}[V]z) \text{ else } \text{ERROR}$

GT is a semantic function that maps the alphanumeric representation of a term in the syntactic category DELT A TERM onto its corresponding term in the semantic domain DERIVATION TERM , if it denotes a valid term in a boolean predicate for the derivation operator δ and a given signature. Otherwise, GT maps the term onto ERROR .

$\text{GT} : [\text{DELT A TERM} \rightarrow \text{SIGNATURE}] \rightarrow [\text{DERIVATION TERM} + \{\text{ERROR}\}]$

$\text{GT}[GF_1 \text{ RO } GF_2]z =$

if $(\text{VALIDGF}[GF_1]z \wedge \text{VALIDGF}[GF_2]z)$

then $\text{GF}[GF_1]z \text{ RO}[RO] \text{ GF}[GF_2]z$

else ERROR

$\text{GT}[V_1 = V_2]z =$

if $(\text{VALIDTE}[V_1]z \wedge \text{VALIDTE}[V_2]z)$

then $\text{TE}[V_1]z = \text{TE}[V_2]z$

else ERROR

H is a semantic function that maps each alphanumeric representation of a historical state in the syntactic category $\mathcal{H}\text{-STATE}$ onto its corresponding historical state in the semantic domain HISTORICAL STATE , if it denotes a valid historical state on a given signature. Otherwise, H maps the historical state onto ERROR .

$H : [H\text{-STATE} \rightarrow SIGNATURE] \rightarrow [HISTORICAL\ STATE + \{ERROR\}]$

$H[\epsilon]z = \emptyset$

$H[HT]z = \text{if } HTUPLE[HT]z = ht \text{ then } \{ht\} \text{ else } \text{ERROR}$

$H[HT_1, HT_2 \dots]z =$

if $(HTUPLE[HT_1]z = ht \wedge H[HT_2 \dots]z = h$
 $\wedge \exists I, (I \in IDENTIFIER \wedge z(I) \neq \text{UNBOUND} \wedge \text{Valid}(ht(I)) \neq \emptyset)$
 $\wedge \forall ht', ht' \in h, \exists I, (I \in IDENTIFIER \wedge z(I) \neq \text{UNBOUND}$
 $\wedge \text{Value}(ht'(I)) \neq \text{Value}(ht(I))))$

then $h \cup \{ht\}$

else ERROR

HTUPLE is a semantic function that maps each alphanumeric representation of a historical tuple in the syntactic category $H\text{-TUPLE}$ onto its corresponding historical tuple in the semantic domain $HISTORICAL\ TUPLE$, if it denotes a valid historical tuple on a given signature. Otherwise, **HTUPLE** maps the tuple onto **ERROR**.

$HTUPLE : [H\text{-TUPLE} \rightarrow SIGNATURE] \rightarrow [HISTORICAL\ TUPLE + \{ERROR\}]$

$HTUPLE[(I : S \in TS)]z =$

if $(z(I) = D_x \wedge D_x[S] \neq \text{ERROR} \wedge TS[TS] \neq \text{ERROR}$
 $\wedge \forall I', I' \in IDENTIFIER \wedge I' \neq I, z(I') = \text{UNBOUND})$

then $\{(I, (D_x[S], TS[TS]))\}$

else ERROR

$HTUPLE[(I_1 : S_1 \in TS_1, I_2 : S_2 \in TS_2 \dots)]z =$

if $(z(I_1) = D_x \wedge D_x[S_1] \neq \text{ERROR} \wedge TS[TS_1] \neq \text{ERROR}$
 $\wedge HTUPLE[(I_2 : S_2 \in TS_2 \dots)](z - \{(I_1, D_x)\} \cup \{(I_1, \text{UNBOUND})\}) = ht)$

then $ht \cup \{(I_1, (D_x[S_1], TS[TS_1]))\}$

else ERROR

N is a semantic function that maps the syntactic category $NUMERAL$ of decimal numerals into the semantic domain $INTEGER$ of integers.

$N : NUMERAL \rightarrow INTEGER$

$$N[0] = 0$$

$$N[1] = 1$$

$$N[2] = 2$$

...

R is a semantic function that maps an expression onto the set of identifiers in the expression that name relations.

$R : EXPRESSION \rightarrow \wp(IDENTIFIER)$

$$R[[snapshot, Z, S]] = \emptyset$$

$$R[[historical, Z, H]] = \emptyset$$

$$R[I] = \{I\}$$

$$R[E_1 \cup E_2] = R[E_1] \cup R[E_2]$$

$$R[E_1 - E_2] = R[E_1] \cup R[E_2]$$

$$R[E_1 \times E_2] = R[E_1] \cup R[E_2]$$

$$R[\sigma F(E)] = R[E]$$

$$R[\pi X(E)] = R[E]$$

$$R[\rho(I, N)](d, tn) = \{I\}$$

$$R[E_1 \dot{\cup} E_2] = R[E_1] \cup R[E_2]$$

$$R[E_1 \dot{-} E_2] = R[E_1] \cup R[E_2]$$

$$R[E_1 \dot{\times} E_2] = R[E_1] \cup R[E_2]$$

$$R[\delta F(E)] = R[E]$$

$$R[\pi X(E)] = R[E]$$

$$R[\delta X(E)] = R[E]$$

$$R[\rho(I, N)](d, tn) = \{I\}$$

$$E[\widehat{A} I_1, W, I_2, I_3, B(E_1, E_2)] = R[E_1] \cup R[E_2]$$

$$E[\widehat{AU} I_1, W, I_2, I_3, B(E_1, E_2)] = R[E_1] \cup R[E_2]$$

RO is a semantic function that maps each alphanumeric representation of a relational operator in the syntactic category $REL\ OP$ onto the relational operator that it denotes in the semantic domain $RELATIONAL\ OPERATOR$.

$\text{RO} : \text{REL OP} \rightarrow \text{RELATIONAL OPERATOR}$

$\text{RO}[<] = <$

$\text{RO}[=] = =$

$\text{RO}[>] = >$

S is a semantic function that maps each alphanumeric representation of a snapshot state in the syntactic category $S\text{-STATE}$ onto its corresponding snapshot state in the semantic domain $S\text{/SNAPSHOT STATE}$, if it denotes a valid snapshot state on a given signature. Otherwise, S maps the snapshot state onto ERROR .

$S : [S\text{-STATE} \rightarrow \text{SIGNATURE}] \rightarrow [SNAPSHOT STATE + \{\text{ERROR}\}]$

$S[\epsilon]z = \emptyset$

$S[ST]z = \text{if STUPLE}[ST]z = st \text{ then } \{st\} \text{ else } \text{ERROR}$

$S[ST_1, ST_2 \dots]z =$

if $(\text{STUPLE}[ST_1]z = st \wedge S[ST_2 \dots]z = s)$
 $\wedge \forall st', st' \in s, \exists I, (I \in IDENTIFIER \wedge z(I) \neq \text{UNBOUND}$
 $\wedge st'(I) \neq st(I))$

then $s \cup \{st\}$

else ERROR

SO is a semantic function that maps each alphanumeric representation of a set operator in the syntactic category $SET\text{ OP}$ onto the set operator that it denotes in the semantic domain $SET\text{ OPERATOR}$.

$SO : SET\text{ OP} \rightarrow SET\text{ OPERATOR}$

$SO[\cup] = \cup$

$SO[-] = -$

$SO[\cap] = \cap$

STUPLE is a semantic function that maps each alphanumeric representation of a snapshot tuple in the syntactic category $S\text{-TUPLE}$ onto its corresponding snapshot tuple in the

semantic domain $S\text{-TUPLE}$, if it denotes a valid snapshot tuple on a given signature. Otherwise, STUPLE maps the tuple onto ERROR .

$\text{STUPLE} : [S\text{-TUPLE} \rightarrow \text{SIGNATURE}] \rightarrow [S\text{-TUPLE} + \{\text{ERROR}\}]$

$\text{STUPLE}[(I : S)]z =$

```
if      ( $z(I) = D_x \wedge D_x[S] \neq \text{ERROR}$ 
        $\wedge \forall I', I' \in IDENTIFIER \wedge I' \neq I, z(I') = \text{UNBOUND}$ )
then   $\{(I, D_x[S])\}$ 
else    $\text{ERROR}$ 
```

$\text{STUPLE}[(I_1 : S_1, I_2 : S_2 \dots)]z =$

```
if      ( $z(I_1) = D_x \wedge D_x[S_1] \neq \text{ERROR}$ 
        $\wedge \text{STUPLE}[(I_2 : S_2 \dots)](z - \{(I_1, D_x)\} \cup \{(I_1, \text{UNBOUND})\}) = st$ )
then   $st \cup \{(I_1, D_x[S_1])\}$ 
else    $\text{ERROR}$ 
```

TE is a semantic function that maps the alphanumeric representation of a temporal expression in the syntactic category $TIME\ EXPRESSION$ onto its corresponding temporal expression in the semantic domain $TEMPORAL\ EXPRESSION$, if it denotes a valid temporal expression for the derivation operator δ and a given signature. Otherwise, TE maps the expression onto ERROR .

$\text{TE} : [TIME\ EXPRESSION \rightarrow \text{SIGNATURE}] \rightarrow$

$[TEMPORAL\ EXPRESSION + \{\text{ERROR}\}]$

```

TE[I] z = if z(I) =  $\mathcal{D}_x$  then I else ERROR

TE[TS] z = TS[TS] z

TE[EXTEND(GF1, GF2)] z =
    if (VALIDGF[GF1] z  $\wedge$  VALIDGF[GF2] z)
    then Extend(GF[GF1] z, GF[GF2] z)
    else ERROR

TE[V1 SO V2] z =
    if (VALIDTE[V1] z  $\wedge$  VALIDTE[V2] z)
    then TE[V1] z SO[SO] TE[V2] z
    else ERROR

TE[(V)] z = if VALIDTE[V] z then (TE[V] z) else ERROR

```

TS is a semantic function that maps each alphanumeric representation of a set of time quanta in the syntactic category *TIME SET* onto its corresponding set of time quanta in the semantic domain $\wp(T)$.

TS : *TIME SET* \rightarrow $\wp(T)$

```

TS[all] = T
TS[{ }] =  $\emptyset$ 
TS[{T}] = {N[T]}
TS[{T1, T2...}] = {N[T1]}  $\cup$  TS[{T2...}]

```

V is a semantic function that maps the alphanumeric representation of a set of assignments in the syntactic category *TIME LIST* onto its corresponding set of ordered pairs in the semantic domain $\wp(\text{IDENTIFIER} \times \text{TEMPORAL EXPRESSION})$, if all the assignments denote valid pairs of attributes and temporal expressions for the derivation operator δ and a given signature. Otherwise, **V** maps the assignment onto **ERROR**.

```

V : [ TIME LIST  $\rightarrow$  SIGNATURE ]  $\rightarrow$ 
[  $\wp(\text{IDENTIFIER} \times \text{TEMPORAL EXPRESSION}) + \{\text{ERROR}\}$  ]

```

```

V[(I := V)] z =
  if      (z(I) =  $\mathcal{D}_x$   $\wedge$  TE[V]  $\neq$  ERROR
           $\wedge \forall I', I' \in IDENTIFIER \wedge I' \neq I, z(I') = UNBOUND$ )
  then  {(I, TE[V])}
  else   ERROR

V[(I1 := V1, I2 := V2...)] z =
  if      (z(I1) =  $\mathcal{D}_x$   $\wedge$  TE[V1]  $\neq$  ERROR
           $\wedge V[(I_2 := V_2...)](z - \{(I_1, \mathcal{D}_x)\} \cup \{(I_1, UNBOUND)\}) \neq ERROR$ )
  then  V[(I2 := V2...)](z - {(I1,  $\mathcal{D}_x$ )})  $\cup$  {(I1, TE[V1])})
  else   ERROR

```

VALIDB is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *BY LIST* onto the boolean value **TRUE** or **FALSE**, to indicate whether the identifiers denote a valid subset of the attributes in a given signature.

VALIDB : [*BY LIST* \rightarrow *SIGNATURE*] \rightarrow { **TRUE**, **FALSE** }

```

VALIDB[] z = TRUE
VALIDB[(I)] z = (z(I) =  $\mathcal{D}_x$ )
VALIDB[(I1, I2...)] z =
  (z(I1) =  $\mathcal{D}_x$   $\wedge$  VALIDB[(I2...)](z - {(I1,  $\mathcal{D}_x$ )})  $\cup$  {(I1, UNBOUND)})

```

VALIDF is a semantic function that maps the alphanumeric representation of a boolean predicate in the syntactic category *SIGMA EXPRESSION* onto the boolean value **TRUE** or **FALSE**, to indicate whether the predicate is a valid boolean predicate for the selection operator σ (or $\hat{\sigma}$) and a given signature.

VALIDF : [*SIGMA EXPRESSION* \rightarrow *SIGNATURE*] \rightarrow { **TRUE**, **FALSE** }

VALIDF[FT]z = VALIDFT[FT]z

VALIDF[F₁ and F₂]z = (VALIDF[F₁]z \wedge VALIDF[F₂]z)

VALIDF[F₁ or F₂]z = (VALIDF[F₁]z \wedge VALIDF[F₂]z)

VALIDF[not F]z = VALIDF[F]z

VALIDF[(F)]z = VALIDF[F]z

VALIDFT is a semantic function that maps the alphanumeric representation of a term in the syntactic category *SIGMA TERM* onto the boolean value **TRUE OR FALSE**, to indicate whether the term is a valid term in a boolean predicate for the selection operator σ (or $\hat{\sigma}$) and a given signature.

VALIDFT : [SIGMA TERM \rightarrow SIGNATURE] \rightarrow { TRUE, FALSE }

VALIDFT[I₁ RO I₂]z = (z(I₁) = z(I₂) = D_x)

VALIDFT[I RO S]z = (z(I) = D_x \wedge D_x[S] \neq ERROR)

VALIDFT[S RO I]z = (z(S) = D_x \wedge D_x[I] \neq ERROR)

VALIDG is a semantic function that maps the alphanumeric representation of a temporal predicate in the syntactic category *DELTA EXPRESSION* onto the boolean value **TRUE OR FALSE**, to indicate whether the predicate is a valid boolean predicate for the derivation operator δ and a given signature.

VALIDG : [DELTA EXPRESSION \rightarrow SIGNATURE] \rightarrow { TRUE, FALSE }

VALIDG[GT]z = VALIDGT[GT]z

VALIDG[G₁ and G₂]z = (VALIDG[G₁]z \wedge VALIDG[G₂]z)

VALIDG[G₁ or G₂]z = (VALIDG[G₁]z \wedge VALIDG[G₂]z)

VALIDG[not G]z = VALIDG[G]z

VALIDG[(G)]z = VALIDG[G]z

VALIDGF is a semantic function that maps the alphanumeric representation of a factor in the syntactic category *DELTA FACTOR* onto the boolean value **TRUE OR FALSE**, to indicate whether the factor is a valid factor in a temporal predicate for the delta operator δ and a given signature.

VALIDGF : [*DELT A FACTOR* \rightarrow *SIGNATURE*] \rightarrow { TRUE, FALSE }

VALIDGF[*T*] $z = \text{TRUE}$

VALIDGF[*FIRST(V)*] $z = \text{VALIDTE}[V] z$

VALIDGF[*LAST(V)*] $z = \text{VALIDTE}[V] z$

VALIDGT is a semantic function that maps the alphanumeric representation of a term in the syntactic category *DELT A TERM* onto the boolean value TRUE OR FALSE, to indicate whether the term is a valid term in a temporal predicate for the delta operator δ and a given signature.

VALIDGT : [*DELT A TERM* \rightarrow *SIGNATURE*] \rightarrow { TRUE, FALSE }

VALIDGT[*GF₁ RO GF₂*] $z = (\text{VALIDGF}[GF_1] z \wedge \text{VALIDGF}[GF_2] z)$

VALIDGT[*V₁ = V₂*] $z = (\text{VALIDTE}[V_1] z \wedge \text{VALIDTE}[V_2] z)$

VALIDTE is a semantic function that maps the alphanumeric representation of a temporal expression in the syntactic category *TIME EXPRESSION* onto the boolean value TRUE OR FALSE, to indicate whether the expression is a valid temporal expression for the derivation operator δ and a given signature.

VALIDTE : [*TIME EXPRESSION* \rightarrow *SIGNATURE*] \rightarrow { TRUE, FALSE }

VALIDTE[*I*] $z = (z(I) \neq \text{UNBOUND})$

VALIDTE[*TS*] $z = \text{TRUE}$

VALIDTE[*EXTEND(GF₁, GF₂)*] $z = (\text{VALIDGF}[GF_1] z \wedge \text{VALIDGF}[GF_2] z)$

VALIDTE[*V₁ SO V₂*] $z = (\text{VALIDTE}[V_1] z \wedge \text{VALIDTE}[V_2] z)$

VALIDTE[*(V)*] $z = \text{VALIDTE}[V] z$

VALIDV is a semantic function that maps the alphanumeric representation of a set of assignments in the syntactic category *TIME LIST* to the boolean value TRUE OR FALSE, to indicate whether the assignments denote valid pairs of attributes and temporal functions for the derivation operator δ and a given signature.

VALIDV : [*TIME LIST* \rightarrow *SIGNATURE*] \rightarrow { TRUE, FALSE }

VALIDV[(*I* := *V*)] *z* =

$$(z(I) = \mathcal{D}_x \wedge \text{VALIDTE}[V]) \\ \wedge \forall I', I' \in \text{IDENTIFIER} \wedge I' \neq I, z(I') = \text{UNBOUND})$$

VALIDV[(*I*₁ := *V*₁, *I*₂ := *V*₂...)] *z* =

$$(z(I_1) = \mathcal{D}_x \wedge \text{VALIDTE}[V_1]) \\ \wedge \text{VALIDV}[(I_2 := V_2...)](z - \{(I_1, \mathcal{D}_x)\} \cup \{(I_1, \text{UNBOUND})\}))$$

VALIDW is a semantic function that maps the alphanumeric representation of an aggregation windowing function in the syntactic category *WINDOW FUNCTION* onto the boolean value TRUE or FALSE, to indicate whether the function denotes a member of an arbitrary semantic domain of aggregation windowing functions. We assume that the semantic domain of aggregation windowing functions contains, as a minimum, the constant aggregation windowing functions.

VALIDW : *WINDOW FUNCTION* \rightarrow { TRUE, FALSE }

VALIDW[infinity] = TRUE

VALIDW[*N*] = TRUE

VALIDW[*I*] = (WN[*I*] \neq UNBOUND)

VALIDX is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto the boolean value TRUE or FALSE, to indicate whether the identifiers denote a valid subset of the attributes in a given signature.

VALIDX : [*IDENTIFIER LIST* \rightarrow *SIGNATURE*] \rightarrow { TRUE, FALSE }

VALIDX[()] *z* = TRUE

VALIDX[(*I*)] *z* = (z(*I*) = \mathcal{D}_x)

VALIDX[(*I*₁, *I*₂...)] *z* =

$$(z(I_1) = \mathcal{D}_x \wedge \text{VALIDX}[(I_2...)](z - \{(I_1, \mathcal{D}_x)\} \cup \{(I_1, \text{UNBOUND})\}))$$

W is a semantic function that maps the alphanumeric representation of an aggregation windowing function in the syntactic category *WINDOW FUNCTION* onto an element in the arbitrary semantic domain *AGGREGATION WINDOW FUNCTION*, if the function denotes a member of this semantic domain. Otherwise, W maps the function onto *ERROR*. We assume that the semantic domain of aggregation windowing functions contains, as a minimum, the constant aggregation windowing functions.

$$W : \text{WINDOW FUNCTION} \rightarrow [\text{AGGREGATION WINDOW FUNCTION} + \{ \text{ERROR} \}]$$

$$W[\text{infinity}] = \infty$$

$$W[N] = N[N]$$

$$W[I] = \text{if } WN[I] \neq \text{UNBOUND} \text{ then } WN[I] \text{ else } \text{ERROR}$$

X is a semantic function that maps the alphanumeric representation of a list of identifiers in the syntactic category *IDENTIFIER LIST* onto an element in $\wp(\text{IDENTIFIER})$, the power set of *IDENTIFIER*, if the identifiers denote a valid subset of the attributes in a given signature. Otherwise, X maps the list onto *ERROR*.

$$X : [\text{IDENTIFIER LIST} \rightarrow \text{SIGNATURE}] \rightarrow [\wp(\text{IDENTIFIER}) + \{ \text{ERROR} \}]$$

$$X[(\)]z = \emptyset$$

$$X[(I)]z = \text{if } z(I) = D_x \text{ then } \{I\} \text{ else } \text{ERROR}$$

$$X[(I_1, I_2 \dots)]z =$$

$$\begin{aligned} & \text{if } (z(I_1) = D_x \\ & \quad \wedge X[(I_2 \dots)](z - \{(I_1, D_x)\} \cup \{(I_1, \text{UNBOUND})\}) \neq \text{ERROR}) \\ & \text{then } \{I\} \cup X[(I_2 \dots)](z - \{(I_1, D_x)\} \cup \{(I_1, \text{UNBOUND})\}) \\ & \text{else } \text{ERROR} \end{aligned}$$

Y is a semantic function that maps each character string in the syntactic category *CLASS* onto the relation class that it denotes in the semantic domain *RELATION CLASS*.

$$Y : \text{CLASS} \rightarrow \text{RELATION CLASS}$$

$Y[\text{snapshot}] = \text{SNAPSHOT}$
 $Y[\text{rollback}] = \text{ROLLBACK}$
 $Y[\text{historical}] = \text{HISTORICAL}$
 $Y[\text{temporal}] = \text{TEMPORAL}$

Y' is the same as the semantic function Y with the exception that it maps the special symbol * onto a relation's current class.

$Y' : [[\text{CLASS} + \{\ast\}]] \rightarrow \text{RELATION} \rightarrow \text{RELATION CLASS}$

$Y'[\ast](u, v, w) = \text{LASTCLASS}((u, v, w))$
 $Y'[\text{snapshot}](u, v, w) = \text{SNAPSHOT}$
 $Y'[\text{rollback}](u, v, w) = \text{ROLLBACK}$
 $Y'[\text{historical}](u, v, w) = \text{HISTORICAL}$
 $Y'[\text{temporal}](u, v, w) = \text{TEMPORAL}$

Z is a semantic function that maps each alphanumeric representation of a relational signature in the syntactic category SIGNATURE onto its corresponding relational signature in the semantic domain $\text{RELATION SIGNATURE}$, if it denotes a valid signature for the mapping DN from identifiers (i.e., domain names) to value domains. Otherwise, Z maps the signature onto ERROR .

$Z : \text{SIGNATURE} \rightarrow [\text{RELATION SIGNATURE} + \{\text{ERROR}\}]$

$Z[(I_{1,1} : I_{1,2})] =$
 if $\text{DN}[I_{1,2}] \neq \text{UNBOUND}$
 then $\{(I_{1,1}, \text{DN}[I_{1,2}])\} \cup \{(I, \text{UNBOUND}) \mid I \in \text{IDENTIFIER} \wedge I \neq I_{1,1}\}$
 else ERROR
 $Z[(I_{1,1} : I_{1,2}, I_{2,1} : I_{2,2} \dots)] =$
 if $Z[(I_{2,1} : I_{2,2} \dots)] = z \wedge z(I_{1,1}) = \text{UNBOUND} \wedge \text{DN}[I_{1,2}] \neq \text{UNBOUND}$
 then $z - \{(I_{1,1}, \text{UNBOUND})\} \cup \{(I_{1,1}, \text{DN}[I_{1,2}])\}$
 else ERROR

Z' is the same as the semantic function Z with the exception that it maps the special symbol * onto a relation's current signature.

$$Z' : [[\text{SIGNATURE} + \{*\}] \rightarrow \text{RELATION}] \rightarrow [\text{RELATION SIGNATURE} + \{ \text{ERROR} \}]$$

$$Z'[*](u, v, w) = \text{LASTSIGNATURE}((u, v, w))$$

$$Z'[(I_{1,1} : I_{1,2})](u, v, w) =$$

if $\text{DN}[I_{1,2}] \neq \text{UNBOUND}$

then $\{(I_{1,1}, \text{DN}[I_{1,2}])\} \cup \{(I, \text{UNBOUND}) \mid I \in \text{IDENTIFIER} \wedge I \neq I_{1,1}\}$

else ERROR

$$Z'[(I_{1,1} : I_{1,2}, I_{2,1} : I_{2,2} \dots)](u, v, w) =$$

if $Z[(I_{2,1} : I_{2,2} \dots)] = z \wedge z(I_{1,1}) = \text{UNBOUND} \wedge \text{DN}[I_{1,2}] \neq \text{UNBOUND}$

then $z - \{(I_{1,1}, \text{UNBOUND})\} \cup \{(I_{1,1}, \text{DN}[I_{1,2}])\}$

else ERROR

B.2 Other Auxiliary Functions

In addition to the auxiliary semantic functions used in the definitions of expressions and commands in Chapter 4, several other auxiliary functions appear in Chapters 3 and 4. We present here formal definitions for those functions, along with formal definitions for all functions used, in turn, in their definitions. For these definitions, let

d range over the domain $\mathcal{D} = \{\mathcal{D}_1, \dots, \mathcal{D}_e\}$,

h range over the domain HISTORICAL STATE ,

ht range over the domain HISTORICAL TUPLE ,

I range over the syntactic category IDENTIFIER ,

IN range over the domain \mathcal{IN} of intervals defined on page 299,

l , l_1 , and l_2 range over the domain $\text{SNAPSHOT STATE} + \text{HISTORICAL STATE}$,

s range over the domain SNAPSHOT STATE ,

st range over the domain SNAPSHOT TUPLE ,

T range over the domain $\mathcal{P}(T)$, the power set of the domain T ,

t , t' , t_1 , t_2 , t_P , and t_S range over the domain T ,

tn , tn' , tn_1 , and tn_2 range over the domain $[\text{TRANSACTION NUMBER} + \{-\}]$,

u and u' range over the domain $[\text{RELATION CLASS} \times \text{TRANSACTION NUMBER}]^*$,

v and v' range over the domain

$[RELATION SIGNATURE \times TRANSACTION NUMBER]^*$,

w and w' range over the domain

$[[SNAPSHOT STATE \times TRANSACTION NUMBER] +$

$[HISTORICAL STATE \times TRANSACTION NUMBER]]^*$,

y, y_1 , and y_2 range over the domain $RELATION CLASS$; and

z, z_1 , and z_2 range over the domain $RELATION SIGNATURE$.

Again, some of these conflict with the usage as given in Appendix A; such conflict was unavoidable.

BaseRelation determines whether an identifier denotes a defined base relation in a database state. For this function's definition, assume that relations are elements of the semantic domain $RELATION$ as defined on page 154.

BaseRelation :

$[IDENTIFIER \times DATABASE STATE] \rightarrow \{TRUE, FALSE\}$

BaseRelation(I, d) =

$d(I) = (u_1, u_2, u_3, BASE) \wedge LastClass(d(I)) \neq \text{UNDEFINED}$

Close maps a relation's class sequence u and a transaction number tn onto the subsequence recorded through transaction tn . It also sets the second transaction-number component in the last element of the resulting sequence to tn if the component is either “-” or greater than tn .

Close :

$[[RELATION CLASS \times TRANSACTION NUMBER \times$

$[TRANSACTION NUMBER + \{-\}]]^* \times TRANSACTION NUMBER] \rightarrow$

$[RELATION CLASS \times TRANSACTION NUMBER \times$

$[TRANSACTION NUMBER + \{-\}]]^*$

```

Close(u, tn) =
  if   (u ≠ () ∧ Head(u) = (y, tn1, tn2) ∧ tn1 ≤ tn)
    then if   Tail(u) ≠ ()
      then  ⟨ Head(u) ⟩ || Close(Tail(u), tn)
      else if (tn2 = - ∨ (tn2 ≠ - ∧ tn2 > tn))
        then  ⟨ (y, tn1, tn) ⟩
        else   ⟨ (y, tn1, tn2) ⟩
    else   ⟨ ⟩

```

where *Head* and *Tail* are the head and tail operations for sequences and “||” is the concatenation operator on sequences.

Consistent is a boolean function that determines whether a class and signature are consistent with an expression's type.

Consistent :

$$\begin{aligned} & [\text{RELATION CLASS} \times \text{RELATION SIGNATURE} \times \\ & [\text{RELATION CLASS} \times \text{RELATION SIGNATURE}]] \rightarrow \{ \text{TRUE}, \text{FALSE} \} \end{aligned}$$

```

Consistent(y1, z1, (y2, z2)) =
  ((z1 = z2) ∧ ( ( y1 = SNAPSHOT ∧ y2 = SNAPSHOT )
    ∨ ( y1 = ROLLBACK ∧ y2 = SNAPSHOT )
    ∨ ( y1 = HISTORICAL ∧ y2 = HISTORICAL )
    ∨ ( y1 = TEMPORAL ∧ y2 = HISTORICAL ) ))

```

Expand replaces the second transaction-number component in the last element of a relation's MSoT class sequence with the special element “-”.

Expand : [RELATION + { ((), (), ()) }] → [RELATION + { ((), (), ()) }]

```

Expand((u, v, w) ) =
  if      u ≠ ⟨ ⟩
  then   if      (Tail(u) ≠ ⟨ ⟩ ∧ Expand((Tail(u), v, w) ) = (u', v', w') )
         then ((⟨(y1, tn1, tn2)⟩ || u', v, w)
         else  ((⟨(y1, tn1, -)⟩), v, w)
  else    (u, v, w)

```

where *Head*(*u*) = (*y*, *tn*₁, *tn*₂).

Extend maps two times onto the set of times that represents the interval between the first time and the second time.

Extend : *T* × *T* → [*IN* + {**ERROR**}]

```

Extend(t1, t2) =
  if      t1 ≤ t2
  then   {t | t1 ≤ t ≤ t2}
  else    ERROR

```

FindClass maps a relation onto the class component of the element in the relation's class sequence whose first transaction-number component is less than or equal to a given transaction number and whose second transaction-number component is greater than or equal to the transaction number. If no such element exists in the sequence, then *FindClass* returns **ERROR**.

FindClass :

$$[[\text{RELATION} + \{ ((), (), ()) \}] \times \text{TRANSACTION NUMBER}] \rightarrow [\text{RELATION CLASS} + \{ \text{ERROR} \}]$$

```

FindClass((u, v, w), tn) =
  if      (u ≠ ⟨ ⟩ ∧ tn1 ≤ tn)
  then   if      (tn2 = - ∨ tn ≤ tn2)
         then y
         else  FindClass((Tail(u), v, w), tn)
  else    ERROR

```

where *Head*(*u*) = (*y*, *tn*₁, *tn*₂).

FindSignature maps a relation onto the signature component of the element in the relation's signature sequence having the largest transaction-number component less than or equal to a given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindSignature* returns ERROR.

FindSignature :

$$[[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \times \text{TRANSACTION NUMBER}] \rightarrow [\text{RELATION SIGNATURE} + \{ \text{ERROR} \}]$$

FindSignature((u, v, w), tn) =

```

if      (FindClass(( $u, v, w$ ),  $tn$ ) ≠ ERROR ∧  $v \neq \langle \rangle \wedge tn_1 \leq tn$ )
then   if      (Tail( $v$ ) =  $\langle \rangle \vee (\text{Tail}(v) \neq \langle \rangle \wedge tn < tn_2)$ )
        then  $z_1$ 
        else FindSignature(( $u, \text{Tail}(v), w$ ),  $tn$ )
else   ERROR

```

where *Head*(v) = (z_1, tn_1) and *Head*(*Tail*(v)) = (z_2, tn_2).

FindState maps a relation onto the state component of the element in the relation's state sequence having the largest transaction-number component less than or equal to a given transaction number, if *FindClass* does not return an error for the same transaction number. If *FindClass* returns an error or no such element exists in the sequence, then *FindState* returns ERROR.

FindState :

$$[[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \times \text{TRANSACTION NUMBER}] \rightarrow [\text{SNAPSHOT STATE} + \text{HISTORICAL STATE} + \{ \text{ERROR} \}]$$

FindState((u, v, w), tn) =

```

if      (FindClass(( $u, v, w$ ),  $tn$ ) ≠ ERROR ∧  $w \neq \langle \rangle \wedge tn_1 \leq tn$ )
then   if      (Tail( $w$ ) =  $\langle \rangle \vee (\text{Tail}(w) \neq \langle \rangle \wedge tn < tn_2)$ )
        then  $l_1$ 
        else FindState(( $u, v, \text{Tail}(w)$ ),  $tn$ )
else   ERROR

```

where *Head*(w) = (l_1, tn_1) and *Head*(*Tail*(w)) = (l_2, tn_2).

First takes a set of times from the domain $\wp(T)$ and maps it onto the earliest time in the set.

First : $\wp(T) \rightarrow [T + \{\text{ERROR}\}]$

```

First(T) =
  if   T ≠ ∅
  then t, t ∈ T ∧ ∀t', t' ∈ T, t ≤ t'
  else  ERROR

```

Last takes a set of times from the domain $\wp(T)$ and maps it onto the latest time in the set.

Last : $\wp(T) \rightarrow [T + \{\text{ERROR}\}]$

```

Last(T) =
  if   T ≠ ∅
  then t, t ∈ T ∧ ∀t', t' ∈ T, t ≥ t'
  else  ERROR

```

LastClass maps a relation onto the class component of the last element in the relation's class sequence. If the sequence is empty, *LastClass* returns ERROR.

LastClass :

$[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \rightarrow [\text{RELATION CLASS} + \{\text{ERROR}\}]$

```

LastClass((u, v, w)) =
  if   (u ≠ ⟨ ⟩ ∧ Head(u) = (⟨ y, tn1, tn2 ⟩))
  then if   Tail(u) = ⟨ ⟩
         then y
         else  LastClass((Tail(u), v, w))
  else  ERROR

```

LastSignature maps a relation onto the signature component of the last element in the relation's signature sequence. If the relation's signature sequence is empty, *LastSignature* returns ERROR.

LastSignature :

$[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \rightarrow [\text{RELATION SIGNATURE} + \{\text{ERROR}\}]$

LastSignature((*u*, *v*, *w*) =

```

if      (v ≠ ⟨ ⟩ ∧ Head(v) = (⟨, tn1⟩))
then if      Tail(v) = ⟨ ⟩
      then   z
      else   LastSignature(u, Tail(v), w)
else   ERROR

```

LastState maps a relation onto the state component of the last element in the relation's state sequence. If the relation's state sequence is empty, *LastState* returns ERROR.

LastState :

$[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \rightarrow [\text{SNAPSHOT STATE} + \text{HISTORICAL STATE} + \{\text{ERROR}\}]$

LastState((*u*, *v*, *w*) =

```

if      (w ≠ ⟨ ⟩ ∧ Head(w) = (⟨, tn1⟩))
then if      Tail(w) = ⟨ ⟩
      then   l
      else   LastState(u, v, Tail(w))
else   ERROR

```

LastTrNumber maps a relation's class sequence onto the transaction number of the transaction that appended the last element to the sequence. If the relation's class sequence is empty, *LastTrNumber* returns ERROR.

LastTrNumber :

$[\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times [\text{TRANSACTION NUMBER} + \{-\}]]^* \rightarrow \text{TRANSACTION NUMBER}$

```

LastTrNumber(u) =
  if      (u ≠ () ∧ Head(u) = (y, tn1, tn2))
  then if      Tail(u) = ()
         then   tn1
         else   LastTrNumber(Tail(u))
  else   ERROR

```

Interval maps a set of times onto the set of intervals containing the minimum number of non-disjoint intervals represented by the input set. Each time in the input set appears in exactly one interval in the output set and each interval in the output set is itself represented by a set of times.

Let the domain \mathcal{IN} be the subset of $\mathcal{P}(T)$ that represents all possible non-disjoint intervals of time.

$$\mathcal{IN} \triangleq \{\emptyset\} \cup \{IN \mid IN \in \mathcal{P}(T) \wedge IN \neq \emptyset \wedge \forall t, First(IN) \leq t \leq Last(IN) \rightarrow t \in IN\}$$

Note that \mathcal{IN} includes the empty set and intervals of length 1. Also let $\mathcal{P}(\mathcal{IN})$ be the power set of \mathcal{IN} . While $\mathcal{IN} \subset \mathcal{P}(T)$, each element of $\mathcal{P}(\mathcal{IN})$ is a set, each of whose elements is also an element of $\mathcal{P}(T)$.

Interval : $\mathcal{P}(T) \rightarrow \mathcal{P}(\mathcal{IN})$

```

Interval(T) =
  if      T ≠ ∅
  then {IN | ∀t, t ∈ T →
        (Pred(t) ∈ T → Pred(t) ∈ IN)
        (Succ(t) ∈ T → Succ(t) ∈ IN)}
  else   {∅}

```

MaintenanceStrategy maps an identifier that denotes a view in a database state onto the maintenance strategy for the view. If the identifier does not denote a view, *MaintenanceStrategy* returns **ERROR**. For this function's definition, assume that relations are elements of the semantic domain $\mathcal{RELATION}$ as defined on page 154.

MaintenanceStrategy :

$$[IDENTIFIER \times DATABASE\ STATE] \rightarrow \{\text{UNMATERIALIZED, RECOMPUTED, INCREMENTAL, ERROR}\}$$

```

MaintenanceStrategy( $I, d$ ) = if       $d(I) = (u_1, u_2, u_3, (E, \text{UNMATERIALIZED}))$ 
                                         then  UNMATERIALIZED
                                         else if  $d(I) = (u_1, u_2, u_3, (E, \text{RECOMPUTED}))$ 
                                                 then RECOMPUTED
                                         else if  $d(I) = (u_1, u_2, u_3, (E, \text{INCREMENTAL}))$ 
                                                 then INCREMENTAL
                                         else   ERROR

```

$MSoT$ maps a relation (u, v, w) and a transaction number tn onto the history of the relation as a rollback or temporal relation before the start of transaction tn .

$$MSoT : [[\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}] \times \text{TRANSACTION NUMBER}] \rightarrow [\text{RELATION} + \{(\langle \rangle, \langle \rangle, \langle \rangle)\}]$$

```

MSoT(( $u, v, w$ ),  $tn$ ) =
  if      ( $u' = \text{PrefixClasses}(u, tn) \wedge u' \neq \langle \rangle \wedge tn' = \text{LastTrNumber}(u')$ )
  then if     $\text{MultiStateClass}(\text{LastClass}((u', v, w)))$ 
            then  ( $\text{Close}(u', tn - 1), \text{PrefixSigs}(v, tn), \text{PrefixStates}(w, tn)$ )
            else   ( $\text{PrefixClasses}(u, tn'), \text{PrefixSigs}(v, tn'), \text{PrefixStates}(w, tn')$ )
  else   ( $\langle \rangle, \langle \rangle, \langle \rangle$ )

```

MultiStateClass is a boolean function that determines whether a class is either ROLLBACK or TEMPORAL.

$$\text{MultiStateClass} : \text{RELATION CLASS} \rightarrow \{\text{TRUE}, \text{FALSE}\}$$

$$\text{MultiStateClass}(y) = (y = \text{ROLLBACK} \vee y = \text{TEMPORAL})$$

$NewSignature$ maps a relation's $MSoT$ and a (signature, transaction number) pair onto the empty sequence, if the signature in the last element of the relation's $MSoT$ signature sequence is equal to the signature in the (signature, transaction number) pair, or a one-element sequence containing the (signature, transaction number) pair, otherwise.

NewSignature :

$$\begin{aligned} & [[\text{RELATION} + \{ (\langle \rangle, \langle \rangle, \langle \rangle) \}] \times \\ & [\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]] \rightarrow \\ & [\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^* \end{aligned}$$

NewSignature((u, v, w), (z, tn)) =

```

if      LastSignature((u, v, w)) = z
then   ⟨ ⟩
else   ⟨ (z, tn) ⟩

```

NewState maps a relation's MSoT, a (relation state, transaction number) pair, and a (class, signature) pair onto the empty sequence, if the class and signature in the last elements of the relation's MSoT class and signature sequences are consistent with the (class, signature) pair and the state in the last element of the relation's MSoT state sequence is equal to the relation state in the (relation state, transaction number) pair, or a one-element sequence containing the (relation state, transaction number) pair, otherwise.

NewState :

$$\begin{aligned} & [[\text{RELATION} + \{ (\langle \rangle, \langle \rangle, \langle \rangle) \}] \times \\ & [[\text{SNAPSHOT STATE} + \text{HISTORICAL STATE}] \times \text{TRANSACTION NUMBER}] \times \\ & [\text{RELATION CLASS} \times \text{RELATION SIGNATURE}]] \rightarrow \\ & [[\text{SNAPSHOT STATE} + \text{HISTORICAL STATE}] \times \text{TRANSACTION NUMBER}]^* \end{aligned}$$

NewState((u, v, w), (l, tn), (y, z)) =

```

if      ( Consistent(LastClass((u, v, w)),
          LastSignature((u, v, w)), (y, z)) )
      ^ LastState((u, v, w)) = l
then   ⟨ ⟩
else   ⟨ (l, tn) ⟩

```

Pred is the predecessor function on the domain \mathcal{T} . It maps a time onto its immediate predecessor in the linear ordering of all times.

Pred : $\mathcal{T} \rightarrow [\mathcal{T} + \{ \text{ERROR} \}]$

```

Pred(t) =
    if  $t \neq \text{First}(t)$ 
    then  $t_P, t_P \in T \wedge t_P < t \wedge \forall t', t' \in T \wedge t' < t, t' \leq t_P$ 
    else ERROR

```

PrefixClasses maps a relation's class sequence u and a transaction number tn onto the subsequence recorded before the start of transaction tn .

PrefixClasses :

$$\begin{aligned}
 & [[\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times \\
 & [\text{TRANSACTION NUMBER} + \{-\}]^* \times \text{TRANSACTION NUMBER}] \rightarrow \\
 & [\text{RELATION CLASS} \times \text{TRANSACTION NUMBER} \times \\
 & [\text{TRANSACTION NUMBER} + \{-\}]^*]
 \end{aligned}$$

PrefixClasses(u, tn) =

```

    if  $(u \neq \langle \rangle \wedge \text{Head}(u) = (y, tn_1, tn_2) \wedge tn_1 < tn)$ 
    then  $\langle \text{Head}(u) \rangle \parallel \text{PrefixClasses}(\text{Tail}(u), tn)$ 
    else  $\langle \rangle$ 

```

PrefixSigs maps a relation's signature sequence v and a transaction number tn onto the subsequence recorded before the start of transaction tn .

PrefixSigs :

$$\begin{aligned}
 & [[\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^* \times \\
 & \text{TRANSACTION NUMBER}] \rightarrow \\
 & [\text{RELATION SIGNATURE} \times \text{TRANSACTION NUMBER}]^*
 \end{aligned}$$

PrefixSigs(v, tn) =

```

    if  $(v \neq \langle \rangle \wedge \text{Head}(v) = (z, tn_1) \wedge tn_1 < tn)$ 
    then  $\langle \text{Head}(v) \rangle \parallel \text{PrefixSigs}(\text{Tail}(v), tn)$ 
    else  $\langle \rangle$ 

```

PrefixStates maps a relation's state sequence w and a transaction number tn onto the subsequence recorded before the start of transaction tn .

PrefixStates :

$$\begin{aligned} & [[\text{RELATION STATE} \times \text{TRANSACTION NUMBER}]^* \times \\ & \quad \text{TRANSACTION NUMBER}] \rightarrow \\ & \quad [[\text{SNAPSHOT STATE} + \text{HISTORICAL STATE}] \times \\ & \quad \quad \text{TRANSACTION NUMBER}]^* \end{aligned}$$

PrefixStates(w, tn) =

```

if      ( $w \neq \langle \rangle \wedge \text{Head}(w) = (l, tn_1) \wedge tn_1 < tn$ )
then    $\langle \text{Head}(w) \rangle \parallel \text{PrefixStates}(\text{Tail}(w), tn)$ 
else    $\langle \rangle$ 
```

SingleStateClass is a boolean function that determines whether a class is either **SNAPSHOT** or **HISTORICAL**.

SingleStateClass : *RELATION CLASS* $\rightarrow \{\text{TRUE}, \text{FALSE}\}$

SingleStateClass(y) = ($y = \text{SNAPSHOT} \vee y = \text{ROLLBACK}$)

Succ is the successor function on the domain T . It maps a time onto its immediate successor in the linear ordering of all times.

Pred : $T \rightarrow T$

Succ(t) = $t_s, t_s \in T \wedge t_s > t \wedge \forall t', t' \in T \wedge t' > t, t' \geq t_s$

UpdateState maps a relation state, differential, and relation class onto the relation state that the input relation state and differential denote. If the class is other than snapshot or historical, *UpdateState* returns **ERROR**.

UpdateState :

$$\begin{aligned}
 & [[\text{SNAPSHOT STATE} \times \text{SNAPSHOT DIFFERENTIAL} \times \\
 & \quad \text{RELATION CLASS}] + \\
 & [\text{HISTORICAL STATE} \times \text{HISTORICAL DIFFERENTIAL} \times \\
 & \quad \text{RELATION CLASS}]] - \\
 & [\text{SNAPSHOT STATE} + \text{HISTORICAL STATE} + \{\text{ERROR}\}]
 \end{aligned}$$

UpdateState(l, Δ, y) = if $y = \text{SNAPSHOT}$
 then $S\text{-}_Update(l, \Delta)$
 else if $y = \text{HISTORICAL}$
 then $H\text{-}_Update(l, \Delta)$
 else ERROR

Valid maps an attribute's value in a historical tuple (i.e., a (*value*, *valid*) pair) onto its valid-time component.

Valid : [$\mathcal{D} \times \wp(T)$] $\rightarrow \wp(T)$

$Valid((d, T)) = T$

Value maps an attribute's value in a historical tuple (i.e., a (*value*, *valid*) pair) onto its value component.

Value : [$\mathcal{D} \times \wp(T)$] $\rightarrow \mathcal{D}$

$Value((d, T)) = d$

View determines whether an identifier denotes a view, either unmaterialized or materialized, in a database state. For this function's definition, assume that relations are elements of the semantic domain *RELATION* as defined on page 154.

View :

[*IDENTIFIER* \times *DATABASE STATE*] $\rightarrow \{\text{TRUE}, \text{FALSE}\}$

$\text{View}(I, d) =$

$$\begin{aligned} d(I) &= (u_1, u_2, u_3, (E, \text{UNMATERIALIZED})) \\ \vee d(I) &= (u_1, u_2, u_3, (E, \text{RECOMPUTED})) \\ \vee d(I) &= (u_1, u_2, u_3, (E, \text{INCREMENTAL})) \end{aligned}$$

ViewDef maps an identifier that denotes a view in a database state onto the expression that defines the view. If the identifier does not denote a view, ViewDef returns **ERROR**. For this function's definition, assume that relations are elements of the semantic domain RELATION as defined on page 154.

$\text{ViewDef} :$

$$[\text{IDENTIFIER} \times \text{DATABASE STATE}] \rightarrow [\text{EXPRESSION} + \{\text{ERROR}\}]$$

$$\begin{aligned} \text{ViewDef}(I, d) &= \text{if } (d(I) = (u_1, u_2, u_3, (E, \text{UNMATERIALIZED}))) \\ &\quad \vee d(I) = (u_1, u_2, u_3, (E, \text{RECOMPUTED})) \\ &\quad \vee d(I) = (u_1, u_2, u_3, (E, \text{INCREMENTAL}))) \\ &\text{then } E \\ &\text{else } \text{ERROR} \end{aligned}$$

Views maps an identifier onto the set of identifiers denoting views that depend, either directly or indirectly, on the relation denoted by the identifier in a database state. For this function's definition, assume that relations are elements of the semantic domain RELATION as defined on page 154.

$\text{Views} :$

$$[\text{IDENTIFIER} \times \text{DATABASE STATE}] \rightarrow \wp(\text{IDENTIFIER})$$

$$\begin{aligned} \text{Views}(I, d) &= \{I'' \mid \exists I', (I' \in \text{IDENTIFIER} \\ &\quad \wedge (d(I') = (u_1, u_2, u_3, (E, \text{UNMATERIALIZED}))) \\ &\quad \vee d(I') = (u_1, u_2, u_3, (E, \text{RECOMPUTED}))) \\ &\quad \vee d(I') = (u_1, u_2, u_3, (E, \text{INCREMENTAL})))) \\ &\quad \wedge I \in \mathbf{R}[E] \wedge (I'' = I' \vee I'' \in \text{Views}(I', d))\} \end{aligned}$$

Appendix C

Language Syntax

This appendix describes the syntax of the algebraic language for database query and update defined in Chapters 3 and 4. A variant of Backus-Naur Form (BNF) is used to specify the syntax. Nonterminal symbols appear in *italics*, delimited by “⟨⟩,” and terminal symbols appear in a *typewriter* typeface. In addition to the standard BNF meta-symbols, we use “{ }” to enclose sequences of symbols occurring zero or more times in succession. An ⟨expression⟩ appears within a command and evaluates to a single snapshot or historical state. A ⟨sigma expression⟩ is a boolean expression that appears as the parameter of the selection operator. A ⟨delta expression⟩ and a ⟨time expression⟩ are respectively a boolean expression and a temporal expression; both appear as parameters of the historical operator δ. Such expressions are discussed in detail in Chapter 3.

C.1 Syntax

Shown here is the syntax for the basic language without any of the extensions discussed in Chapter 3.

```
(program) ::= begin_transaction ⟨command⟩ commit_transaction  
| begin_transaction ⟨command⟩ abort_transaction  
| ⟨program⟩ : ⟨program⟩  
  
(command) ::= define_relation(⟨relation name⟩ , ⟨class⟩ , ⟨signature⟩ )  
| modify_relation(⟨relation name⟩ , ⟨classorstar⟩ ,  
                  ⟨signatureorstar⟩ , ⟨expression⟩ )  
| destroy(⟨relation name⟩ )  
| rename_relation(⟨relation name⟩ , ⟨relation name⟩ )  
| ⟨command⟩ , ⟨command⟩
```

$\langle \text{expression} \rangle ::=$

- [snapshot, $\langle \text{signature} \rangle$, $\langle s\text{-state} \rangle$]
- | [historical, $\langle \text{signature} \rangle$, $\langle h\text{-state} \rangle$]
- | $\langle \text{relation name} \rangle$
- | $\langle \text{expression} \rangle \cup \langle \text{expression} \rangle$
- | $\langle \text{expression} \rangle - \langle \text{expression} \rangle$
- | $\langle \text{expression} \rangle \times \langle \text{expression} \rangle$
- | $\pi \langle \text{identifier list} \rangle (\langle \text{expression} \rangle)$
- | $\sigma \langle \text{sigma expression} \rangle (\langle \text{expression} \rangle)$
- | $\rho \langle \text{relation name} \rangle, \langle \text{time constant} \rangle$
- | $\langle \text{expression} \rangle \dot{\cup} \langle \text{expression} \rangle$
- | $\langle \text{expression} \rangle \dot{-} \langle \text{expression} \rangle$
- | $\langle \text{expression} \rangle \dot{\times} \langle \text{expression} \rangle$
- | $\dot{\pi} \langle \text{identifier list} \rangle (\langle \text{expression} \rangle)$
- | $\dot{\sigma} \langle \text{sigma expression} \rangle (\langle \text{expression} \rangle)$
- | $\delta \langle \text{delta expression} \rangle, \langle \text{time list} \rangle (\langle \text{expression} \rangle)$
- | $\widehat{A} \langle \text{agg parameters} \rangle (\langle \text{expression} \rangle, \langle \text{expression} \rangle)$
- | $\widehat{AU} \langle \text{agg parameters} \rangle (\langle \text{expression} \rangle, \langle \text{expression} \rangle)$
- | $\dot{\rho} \langle \text{relation name} \rangle, \langle \text{time constant} \rangle$
- | $(\langle \text{expression} \rangle)$

$\langle \text{signature or star} \rangle ::= \langle \text{signature} \rangle \mid *$

$\langle \text{signature} \rangle ::= (\langle \text{attribute name} \rangle : \langle \text{domain name} \rangle$
 $\quad \quad \quad \{, \langle \text{attribute name} \rangle : \langle \text{domain name} \rangle \})$

$\langle s\text{-state} \rangle ::= \epsilon \mid \langle s\text{-tuple} \rangle \{, \langle s\text{-tuple} \rangle \}$

$\langle s\text{-tuple} \rangle ::= (\langle \text{attribute name} \rangle : \langle \text{string} \rangle \{, \langle \text{attribute name} \rangle : \langle \text{string} \rangle \})$

$\langle h\text{-state} \rangle ::= \epsilon \mid \langle h\text{-tuple} \rangle \{, \langle h\text{-tuple} \rangle \}$

$\langle h\text{-tuple} \rangle ::= (\langle \text{attribute name} \rangle : \langle \text{string} \rangle @ \langle \text{time set} \rangle$
 $\quad \quad \quad \{, \langle \text{attribute name} \rangle : \langle \text{string} \rangle @ \langle \text{time set} \rangle \})$

$\langle \text{identifier list} \rangle ::= () \mid (\langle \text{identifier} \rangle \{ , \langle \text{identifier} \rangle \})$

$\langle \text{sigma expression} \rangle ::= \langle \text{sigma term} \rangle$
 | $\langle \text{sigma expression} \rangle \text{ and } \langle \text{sigma expression} \rangle$
 | $\langle \text{sigma expression} \rangle \text{ or } \langle \text{sigma expression} \rangle$
 | $\text{not } \langle \text{sigma expression} \rangle$
 | $(\langle \text{sigma expression} \rangle)$

$\langle \text{sigma term} \rangle ::= \langle \text{sigma factor} \rangle \langle \text{rel op} \rangle \langle \text{sigma factor} \rangle$

$\langle \text{sigma factor} \rangle ::= \langle \text{attribute name} \rangle \mid \langle \text{string} \rangle$

$\langle \text{rel op} \rangle ::= < \mid = \mid >$

$\langle \text{delta expression} \rangle ::= \text{true} \mid \text{false}$
 | $\langle \text{delta term} \rangle$
 | $\langle \text{delta expression} \rangle \text{ and } \langle \text{delta expression} \rangle$
 | $\langle \text{delta expression} \rangle \text{ or } \langle \text{delta expression} \rangle$
 | $\text{not } \langle \text{delta expression} \rangle$
 | $(\langle \text{delta expression} \rangle)$

$\langle \text{delta term} \rangle ::= \langle \text{delta factor} \rangle \langle \text{rel op} \rangle \langle \text{delta factor} \rangle$
 | $\langle \text{time expression} \rangle = \langle \text{time expression} \rangle$

$\langle \text{delta factor} \rangle ::= \langle \text{time constant} \rangle$
 | $\text{FIRST}(\langle \text{time expression} \rangle)$
 | $\text{LAST}(\langle \text{time expression} \rangle)$

$\langle \text{time list} \rangle ::= (\langle \text{attribute name} \rangle := \langle \text{time expression} \rangle$
 $\quad \quad \quad \{ , \langle \text{attribute name} \rangle := \langle \text{time expression} \rangle \})$

$\langle \text{time expression} \rangle ::= \langle \text{attribute name} \rangle$
 | $\langle \text{time set} \rangle$
 | $\text{EXTEND}(\langle \text{delta factor} \rangle, \langle \text{delta factor} \rangle)$

| $\langle \text{time expression} \rangle \langle \text{set op} \rangle \langle \text{time expression} \rangle$
 | $\langle \langle \text{time expression} \rangle \rangle$

$\langle \text{time set} \rangle ::= \text{all} \mid \{ \langle \text{time sequence} \rangle \}$

$\langle \text{time sequence} \rangle ::= \epsilon \mid \langle \text{time constant} \rangle \{ , \langle \text{time constant} \rangle \}$

$\langle \text{agg parameters} \rangle ::= \langle \text{scalar aggregate} \rangle , \langle \text{window function} \rangle ,$
 $\qquad\qquad\qquad \langle \text{attribute name} \rangle , \langle \text{attribute name} \rangle , \langle \text{by list} \rangle$

$\langle \text{scalar aggregate} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{window function} \rangle ::= \text{infinity} \mid \langle \text{numeral} \rangle \mid \langle \text{identifier} \rangle$

$\langle \text{by list} \rangle ::= () \mid (\langle \text{identifier} \rangle \{ , \langle \text{identifier} \rangle \})$

$\langle \text{set op} \rangle ::= \cup \mid \cap \mid -$

$\langle \text{class or star} \rangle ::= \langle \text{class} \rangle \mid *$

$\langle \text{class} \rangle ::= \text{snapshot} \mid \text{historical} \mid \text{rollback} \mid \text{temporal}$

$\langle \text{relation name} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{attribute name} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{domain name} \rangle ::= \langle \text{identifier} \rangle$

$\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle \{ \langle \text{letter} \rangle \mid \langle \text{digit} \rangle \}$

$\langle \text{string} \rangle ::= " \langle \text{any character other than "} \rangle$
 $\qquad\qquad\qquad \{ \langle \text{any character other than "} \rangle \} "$

$\langle \text{letter} \rangle ::= \text{a} \mid \text{b} \mid \text{c} \mid \text{d} \mid \text{e} \mid \text{f} \mid \text{g} \mid \text{h} \mid \text{i} \mid \text{j} \mid \text{k} \mid \text{l} \mid \text{m}$

| n | o | p | q | r | s | t | u | v | w | x | y | z
| A | B | C | D | E | F | G | H | I | J | K | L | M
| N | O | P | Q | R | S | T | U | V | W | X | Y | Z

(time constant) ::= *(numeral)*

(numeral) ::= *(nonzero digit) { (digit) }*

(digit) ::= 0 | *(nonzero digit)*

(nonzero digit) ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

C.2 Extensions

Shown here is the additional syntax needed for the extensions to the language discussed in Sections 3.3.5, 3.4.4, and 3.6. No syntax for the category *(value expression)* or *(aggregate expression)* is given as these categories may be defined arbitrarily depending on the value domains allowed and the functions on those domains supported.

(expression) ::= | *(expression) ∩ (expression)*
| *(expression) (sigma expression) ⊕ (expression)*
| *(expression) ▷ (expression)*
| *(expression) ÷ (expression)*
| ρ (*(relation name)* , *(time constant)* , *(time constant)*)
| *(expression) ∩ (expression)*
| *(expression) (sigma expression) ⊖ (expression)*
| *(expression) △ (expression)*
| *(expression) ÷ (expression)*
| $\hat{\rho}$ (*(relation name)* , *(time constant)* , *(time constant)*)

(identifier list) ::= | (*(attribute name)* := *(value expression)* @ *(time expression)*)
{ , *(attribute name)* :=
(value expression) @ *(time expression)*) }

(agg parameters) ::= | *(scalar aggregate)* , *(window function)* , *(attribute name)* ,
(attribute name) , *(by list)* , *(value expression)*

{, *(value expression)* }

| *(scalar aggregate)* , *(window function)* ,
 (attribute name) , *(by list)*)
{, *(scalar aggregate)* , *(window function)* ,
 (attribute name) , *(by list)* } ,
(attribute name) , *(aggregate expression)*

Index

This is an index to the definitions of terms and notation used in the paper. As stated in Section 1.6, elements of syntactic categories appear in **fixed-width**, semantic functions appear in **boldface**, and all other functions appear in *Italics* with at least the first letter capitalized.

- AfterImage*, 149
- aggregate**
 - cumulative, 37
 - functions, 36
 - instantaneous, 37
 - non-unique, 40, 45, 71, 76
 - partitioning function, 38
 - scalar, 36
 - unique, 43, 46, 71, 76
 - window function, 37
- aggregates**
 - non-unique, 153
 - unique, 153
- attribute**, 24
- B**, 72, 277
- base relation**, 132
- BaseRelation**, 159, 293
- BeforeImage*, 149
- C**, 77, 158
 - define_incremental_view**, 160
 - define_recomputed_view**, 160
 - define_relation**, 80, 161
 - define_view**, 159
 - destroy**, 86, 164
 - modify_relation**, 83, 162
 - rename_relation**, 87, 165
- cache manager**, 200
- cartesian product**
 - historical, 28, 45, 70, 75
- incremental**
 - historical, 152
 - snapshot, 143
 - snapshot, 68, 74
- chronon**, 3
- class**, 7, 60
- Close**, 293
- Coalesced**, 101
- command**, 57, 77, 153, 158
- concurrency control**
 - scheduler**, 200
 - transaction manager**, 200
- Consistent**, 77, 294
- Countint**, 114
- database**, 3, 62
 - state, 62
 - empty, 91
- define_incremental_view**, 160
- define_recomputed_view**, 160
- define_relation**, 80, 161
- define_view**, 159
- destroy**, 86, 164
- difference**
 - historical, 27, 44, 69, 75
- incremental**
 - historical, 151
 - snapshot, 142
 - snapshot, 67, 74
- differential**, 135
 - historical, 143
 - snapshot, 138
- domain**
 - time, 24
 - value, 24
- E**, 72, 155
 - historical operators**
 - cartesian product**, 75
 - derivation**, 76
 - difference**, 75

non-unique aggregation, 76
projection, 76
rollback, 76
selection, 76
union, 75
unique aggregation, 76
identifier, 73, 155
snapshot operators
 cartesian product, 74
 difference, 74
 projection, 74
 rollback, 74
 selection, 74
 union, 74
state
 historical, 73
 snapshot, 73
E', 157
 identifier, 158
 snapshot
 rollback, 158
 state, 157
 union, 158
evolution
 contents, 52
 scheme, 52
Expand, 78, 294
expression, 57, 71, 155
Extend, 295

F, 72, 278
FindClass, 65, 295
FindSignature, 65, 296
FindState, 72, 296
Firstvalue, 114
First, 296
FT, 278

G, 72, 279
GF, 280
GT, 280

H, 64, 280
HDifference, 151
H-Differential, 144
historical derivation, 34, 45, 70, 76
 incremental, 148
historical operators
 cartesian product, 28, 45, 70, 75
 derivation, 34, 45, 70, 76
 difference, 27, 44, 69, 75

intersection, 46
natural join, 47
non-unique aggregation, 40, 45, 71, 76
projection, 30, 45, 70, 76
quotient, 48
rollback, 71, 76
selection, 29, 45, 70, 76
Θ-join, 47
union, 26, 44, 69
unique aggregation, 43, 46, 71, 76
HProduct, 152
HTUPLE, 281
HUnion, 150
H_Update, 145

identifier, 67, 73, 155, 158
incremental operators
 historical operators
 cartesian product, 152
 derivation, 148
 difference, 151
 non-unique aggregation, 153
 projection, 149
 selection, 148
 union, 150
 unique aggregaton, 153
 snapshot operators
 cartesian product, 143
 difference, 142
 projection, 141
 selection, 141
 union, 142
intersection
 historical, 46
 snapshot, 46
interval, 3
Interval, 299

LastClass, 65, 297
LastSignature, 65, 297
LastState, 73, 298
LastTrNumber, 298
Last, 297

MaintenanceStrategy, 159, 299
modify_relation, 83, 162
MSoT, 77, 300
MultiStateClass, 300

N, 64, 281
natural join

historical, 47
 snapshot, 47
NewSignature, 78, 300
NewState, 78, 301
OrderInt, 115
P, 90
Position, 115
Pred, 301
PrefixClasses, 302
PrefixSigs, 302
PrefixStates, 302
 program, 56, 90
 projection
 historical, 30, 45, 70, 76
 incremental
 historical, 149
 snapshot, 141
 snapshot, 68, 74
 query, 7
 quotient
 historical, 48
 snapshot, 48
R, 159, 282
 recovery
 cache manager, 200
 recovery manager, 200
 recovery manager, 200
 relation, 61, 154
 historical, 5, 23
 rollback, 5
 snapshot, 5
 temporal, 5
rename_relation, 87, 165
 rollback
 historical, 71, 76
 snapshot, 69, 74, 158
RO, 282
S, 64, 283
 scheduler, 200
 scheme, 3, 23, 52
S_Differential, 138
 selection
 historical, 29, 45, 70, 76
 incremental
 historical, 148
 snapshot, 141
 snapshot, 68, 74
 signature, 24, 60
SingleStateClass, 303
Smallest, 116
 snapshot operators
 cartesian product, 68, 74
 difference, 67, 74
 intersection, 46
 natural join, 47
 projection, 68, 74
 quotient, 48
 rollback, 69, 74, 158
 selection, 68, 74
 Θ -join, 46
 union, 67, 74, 158
 snapshots, 136
SO, 283
 state, 24
 database, 62
 historical, 60, 66, 73
 snapshot, 60, 66, 73, 157
STUPLE, 283
Succ, 303
S_Update, 139
T, 64, 154
 historical operators
 cartesian product, 70
 derivation, 70
 difference, 69
 non-unique aggregation, 71
 projection, 70
 rollback, 71
 selection, 70
 union, 69
 unique aggregation, 71
 identifier, 67, 155
 snapshot operators
 cartesian product, 68
 difference, 67
 projection, 68
 rollback, 69
 selection, 68
 union, 67
 state
 historical, 66
 snapshot, 66
TE, 284
 Θ -join
 historical, 47

snapshot, 46
time
 continuous, 3
 discrete, 3
 transaction, 4
 user-defined, 4
 valid, 4
TQuel, 16
 append, 126
 create, 125
 delete, 127
 prototype
 code generator, 180
 interpreter, 182
 replace, 128, 129
 retrieve, 104
 transformation function, 101
transaction manger, 200
transaction number, 60
TS, 285
tuple, 24
type system, 63, 154
Unchanged, 147
union
 historical, 26, 44, 69, 75
 incremental
 historical, 150
 snapshot, 142
 snapshot, 67, 74, 158
update network
 database, 177
 view, 169
UpdateState, 159, 303
UpdateViews, 163

V, 72, 285
VALIDB, 64, 286
VALIDFT, 287
VALIDF, 64, 286
VALIDGF, 287
VALIDGT, 288
VALIDG, 64, 287
VALIDTE, 288
VALIDV, 65
VALIDV, 288
VALIDW, 65
VALIDW, 289
VALIDX, 65
VALIDX, 289

Valid, 304
 value equivalence, 24, 44
Value, 304
VECounterpart, 146
view, 132
 definition, 132, 168
 materialized, 134
 deferred, 134
 immediate, 134
 incremental, 134
 recomputed, 134
 unmaterialized, 134
 in-line, 134
 query modification, 134
View, 159, 304
ViewDef, 159, 305
Views, 159, 305

W, 72, 290

X, 65, 290

Y, 65, 290
Y', 77, 291

Z, 65, 291
Z', 77, 292